Media Engineering and Technology Faculty
German University in Cairo

AR Experiment Designer

Bachelor Thesis

Author: Ahmed Mamdouh Alwasifey
Supervisors: Dr. Wael Abouelsaadat
Submission Date: 01 June, 2023

Media Engineering and Technology Faculty
German University in Cairo

AR Experiment Designer

Bachelor Thesis

Author: Ahmed Mamdouh Alwasifey
Supervisors: Dr. Wael Abouelsaadat
Submission Date: 01 June, 2023

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor Degree

(ii) due acknowlegement has been made in the text to all other material used

---

Ahmed Mamdouh Alwasifey
01 June, 2023

Acknowledgments

In the name of Allah, the Most Gracious, the Most Merciful (Qur'an ,1:1)

I want to express my gratitude to my family for their support and belief in me. In addition, I want to express my appreciation to my friends for all of their support throughout my efforts. Last but not least, I would want to express my gratitude to Dr. Wael Abouelsaadat for his helpfulness and assistance over the entire semester.

Abstract

The use of augmented reality (AR) has become a trend in recent years due to its ability to overlay digital information onto real-world objects and facilitate the explanation of abstract concepts within real-life environments. The purpose of this paper is to develop a general platform, **EduAR**, for teachers to create their science experiments on AR and VR. The platform consists of a designer application that runs on a personal computer and a player application that runs on a headset such as the Oculus Quest. The designer application and the player are described from a user perspective showing the different user interfaces and how to interact with each of them. This thesis also implements the platform showcasing the implementation details and technologies used. Finally, to highlight the capabilities of EduAR, three experiments were explored and presented.

Contents

Chapter 1

Introduction

1.1 Motivation

Experiments are crucial for students to properly understand the topics that they are learning. However, the experiments in STEM schools are fairly limited as teachers often have to limit the scope of their experiments due to hazards such as fire, electricity, or money shortage [18].

Using Augmented Reality (AR) and Virtual Reality (VR) helps navigate this issue by enabling the experiments to be done virtually. This keeps the users safe from any hazards as well as removes the financial concern [10]. Both AR and VR are fairly accessible through headsets that use mobile phones as the driver, such as Google Cardboard. However, a more stable and accurate device could be used such as the Meta Quest 2 and the Meta Quest Pro [14].

1.2 Aim of the project

This thesis aims to develop a general platform for Teachers to create their science experiments on AR and VR. The platform should be extensible and general enough to support the design and running of any type of experiment. The teacher would use a designer application to create their experiment as well as its logic via a visual programming language. Then a VR/AR application would be used in order to play the experiment.

1.3 Thesis Overview

The thesis is organized as follows: First, chapter 2 introduces a background about AR then enabling technologies are described. Then it also delves into some similar projects like DART and Meta-AR.

Next, chapter 3 talks about the system design aim, the user flows, the overall architecture, the database design, and the implementation details of both the designer app and the player. For both of them, an overview is shown and every component is described briefly.

Finally chapter 4 concludes the work of the thesis and shows the limitations of the project, the future work, and potential enhancements to improve the system.

Chapter 2

Background

2.1 Augmented Reality

Augmented Reality (AR) is an innovation that overlays objects generated by computers or information onto the real-world surroundings [2]. This technology creates a new environment that improves the user's experience of reality by fusing the physical and digital worlds. AR has gained significant popularity in recent years, thanks to advancements in mobile devices and software development. AR has various applications in different fields, including education, entertainment, healthcare, and marketing [8].

AR works by using devices such as smartphones, tablets, and smart glasses equipped with AR software and sensors to capture the user's environment and overlay virtual objects on the real world. In order to maintain the virtual objects' proper location and perspective while the user moves around, the technology employs computer vision algorithms to track the user's position and orientation [3].

AR has been applied in various fields, including education, where it has been used to enhance learning experiences. AR technology has enabled students to explore subjects such as history, science, and geography in an interactive way. In healthcare, AR has been used for training medical students, assisting in surgical procedures, and improving rehabilitation programs. In entertainment, AR technology has been used to create interactive games and enhance the user's viewing experience [25].

2.1.1 AR In Education

The education landscape has been transformed by technological advancements, particularly in computer technology, which has opened up new possibilities for creating immersive and authentic learning environments. The integration of various technological tools such as computers, multimedia resources, the internet, e-learning, and mobile devices has proved successful in enhancing existing teaching methods and materials. Augmented

Reality (AR) is a recent addition to this technological arsenal, offering educators a novel approach to teaching by enabling them to overlay digital information onto real-world objects and facilitating the explanation of abstract concepts within real-life environments. [5]

The capacity to visualise abstract and complex concepts that could be challenging to comprehend through conventional teaching techniques is one of the main advantages of AR in STEM education. For instance, AR can be used to create 3D visualizations of scientific phenomena, such as the human body or the solar system, allowing students to explore and interact with these concepts more engagingly and memorably.

Furthermore, AR can also provide opportunities for students to practice and apply their knowledge and skills in real-world contexts, such as through simulations and virtual experiments [1]. By bridging the gap between theory and practice, this can assist students get ready for professions in STEM sectors in the future.

An investigation conducted in Nigeria on achievement in chemistry revealed that involvement in laboratory activities had a noteworthy influence on determining achievement. The study indicated that student attitude towards chemistry, teacher attitude towards laboratory activities, and access to chemistry laboratory materials were the three most crucial factors affecting the variation in achievement, in that order. As laboratories for science subjects are costly facilities, they are not readily available in many settings. [10]

Students can engage with STEM education without worrying about cost or moral considerations, such as the cost of consumables or the potential for animal damage, thanks to the use of AR technology in STEM education. AR allows students to participate in experiments and learn from their mistakes in a secure environment. Tools for virtual, augmented, and mixed augmented reality are used to make this possible. The availability of tools and applications by themselves, however, is insufficient to raise people's levels of education, as experience has demonstrated. Both people and technology must collaborate, but the optimal balance between the two must be determined. [18]

2.2 Enabling Technologies

2.2.1 Unity

Unity is a versatile game engine that operates on multiple platforms and is utilized for the creation of video games, simulations, and various interactive applications [21]. It was first released in 2005 and has since become one of the most popular game engines in the industry. Unity offers a range of features and tools that allow developers to create immersive, high-quality games for multiple platforms, including desktop, mobile, and console [21].

In addition to game development, Unity has gained popularity as a tool for creating Augmented Reality (AR) applications. Unity provides a range of features and tools

specifically designed for AR development, including support for popular AR frameworks such as ARCore and ARKit [21].

Making lifelike 3D models and animations that can be smoothly incorporated into the real environment is one of the key benefits of using Unity for AR development. Unity's physics engine can also be used to simulate real-world interactions between virtual objects and the environment, enhancing the overall user experience [21].

2.2.2 Oculus Quest

Oculus Quest 2 and Oculus Quest Pro are virtual reality (VR) headsets developed by Oculus, a division of Facebook. These headsets offer users a fully immersive VR experience, with 6 degrees of freedom (6DOF) motion tracking, high-resolution displays, and advanced controllers. Additionally, these headsets also have augmented reality (AR) capabilities, which allow users to overlay digital information onto the real world [14].

It could be argued that the Oculus Quest 2 [14] AR device is currently limited to monochrome visuals. Nevertheless, it should be noted that this is only the inaugural iteration of such technology, and future advancements could potentially enhance its capabilities. For instance, the Oculus Quest Pro already features color visualization and with time, production costs are likely to decrease, making it more accessible to a wider audience. This technology has the potential to revolutionize a number of sectors, including gaming, entertainment, healthcare, and education.

(a) Oculus Quest 2 (b) Oculus Quest Pro

Figure 2.1: Oculus Quest Headset [14]

2.3 Previous Work

2.3.1 Meta-AR

Villanueva et al. [22] authored The Meta-AR app which is a collaborative tool intended for classroom use. The application has three types of microtasks, which include visually oriented, knowledge-oriented, and spatially oriented. The visually oriented microtasks can

be done by focusing on a single object, unlike the knowledge-based microtasks that require students to encode information in order to comprehend the instruction. Microtasks that are spatially focused demand motor performance and call for a professional demonstration. The app follows a pull-based development model, which is specially designed to address conflicts among multiple changes, thus creating an efficient collaboration process.

Two types of pull-based models, namely local pull and global pull, are implemented. Local pull allows students to seek help and send pull requests, which other students can add explanatory components to, such as images, videos, and text, and share with their classmates. Students can peruse the suggestions made by contributors and combine the best ones, with changes only affecting their local devices. This process promotes interaction between students and reduces their dependence on instructors. Global pull, on the other hand, requires instructor approval before changes are merged and take effect globally. It is only allowed after students complete a project, and it is handled by the instructor after class. [22]

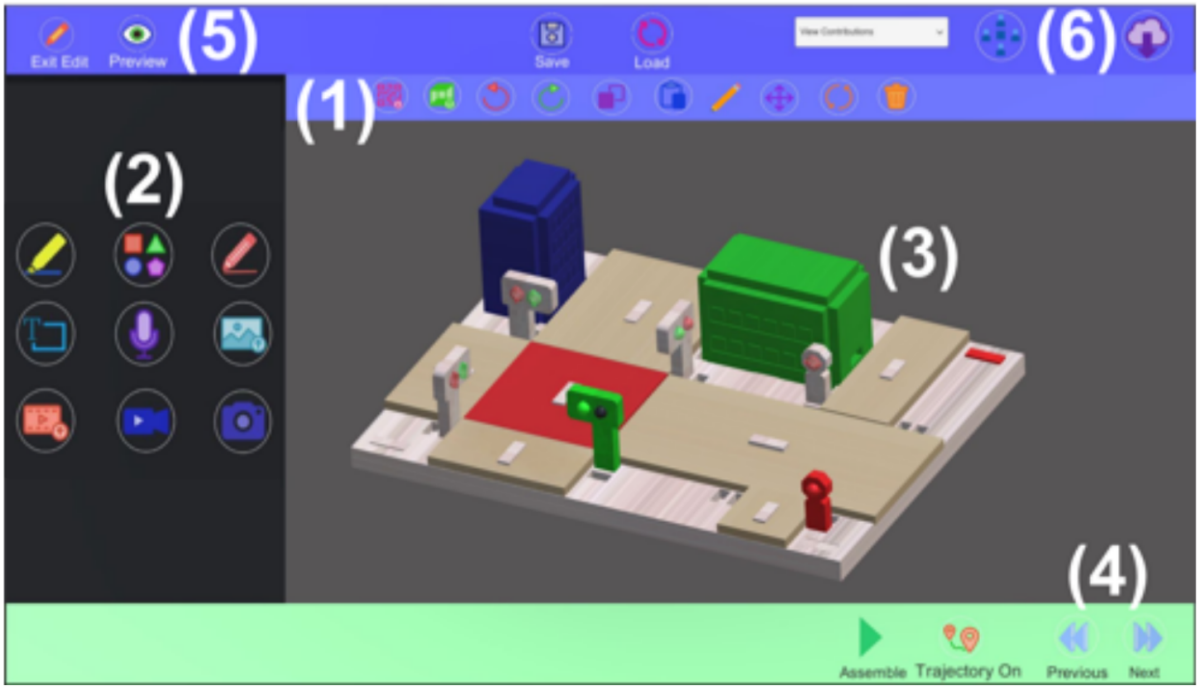


Figure 2.2: Interface of Meta-AR-App [22] in instructor-mode

Meta-AR applications have been designed to meet specific goals, including efficiency accessibility and re-useability. Users can make animations by choosing two points from one object to another using its drag-and-drop interface. Structured XML files are created to store the metadata of each project file inspired by the operating system's file management system. These metadata, which include file types, creator IDs, and file index numbers, act as file controllers and let users track files and carry out other tasks. However, Meta-AR is limited to the creation of animations. [22]

2.3.2 AuthorAR

Lucrecia et al. [12] has created AuthorAR which is an authoring tool for designing educational activities. The tool is specifically targeted toward improving the teaching and learning experiences of students and teachers in the area of special education. It was developed using ActionScript, with each activity generating an XML file that conforms to the tool's schema. AuthorAR consists of an activities generator and a player, with the latter being responsible for activity resolution.

The initial version of AuthorAR allows for the creation of two types of activities: exploratory activities and structuring phrases activities. Exploratory activities can be any activity that helps students learn something new, such as a new concept, rule, formula, or knowledge. On the other hand, structuring phrase templates allow teachers to create activities in which students must compose a phrase in a subject-verb-object way. When the phrase is well-structured, an animation of the subject acting with the corresponding object can be incorporated into the activity. [12]

Despite its potential benefits, the tool has limitations as it only supports a fixed set of activities, rather than a dynamic component-based approach. Nonetheless, the tool can be a valuable resource for educators and students in the area of special education, particularly for improving learning outcomes through the use of multimedia and interactive activities. [12]

2.3.3 DART

MacIntyre et al. [13] created The Designer's AR Toolkit which has been developed through collaborations with designers for the past four years to facilitate their direct and efficient use of AR technology. DART aims to tackle a range of issues that make AR a challenging medium to work with. In order to accomplish this objective, DART is constructed upon the foundation of Macromedia Director, a widely utilized multimedia content creation software that possesses sophisticated debugging and design functionalities. The choice of Director was made because it is a powerful and extensible platform that can be used for final content delivery.

The evolution of media has revealed that the full potential of any medium, including AR, cannot be realized until it is employed by designers who eventually establish the popular forms of the medium through their work. [13]

DART was developed largely to promote the development of modular experiences by defining each actor (piece of content) as a separate entity that is only tangentially related to other actors and to sensing and tracking hardware. [13]

The director application is furnished with an object-oriented programming language called Lingo and is founded upon the analogy of a stage production. The design environment comprises a stage, various casts (housing all elements of content, such as images, videos, 3D content, Lingo scripts, textual data, etc.), a score (depicting the timeline of

the experience), and sprites (which are cast members positioned on the stage or within the score). Lingo scripts, also known as behaviors, are interpreted and can be assigned to cast members, the stage, sprites, or frames within the score. Director generates graphical interfaces for editing behavior properties automatically based on structured comments in the script. [13]

Within Director, the score assumes a pivotal role as the principal organizational element. During runtime, the "play-head" advances from the left side to the right side of the score. As interactive applications do not adhere to a predetermined linear script, developers utilize the score mainly to visually construct discrete logical sections, referred to as frames, through which the program can navigate and iterate. (refer to Figure 2.3 and Figure 2.4). Designers can add behaviors to sprites by dragging behavior scripts onto them and placing sprites on the score. [13]

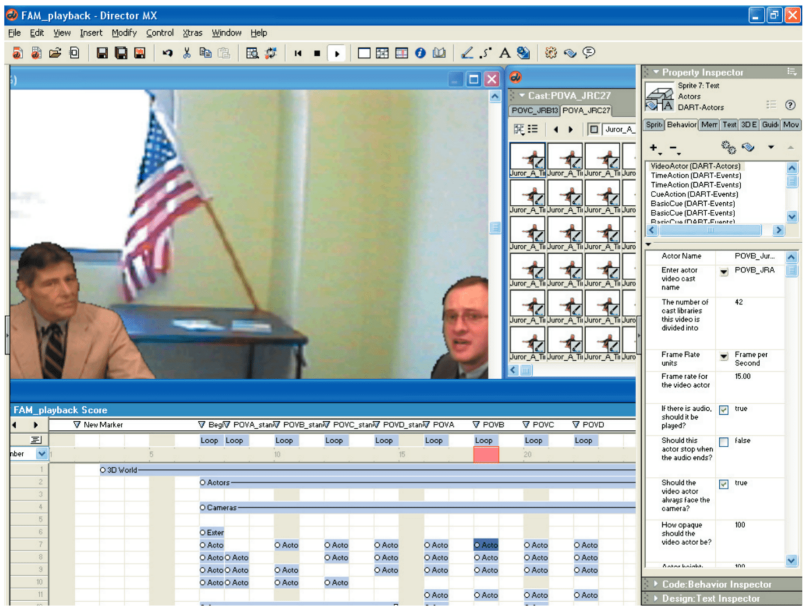


Figure 2.3: DART work session [13]

Figure 2.3 depicts a work session in DART, featuring the complete score for FAM. The composition encompasses nine scenes and multiple actors, with each scene being denoted by a column in the score. The running experience is exhibited on the stage, alongside a partial view of the content belonging to a video actor, as well as certain editing windows within Director.

A basic DART example with the score is shown in Figure 2.4. This frame is repeatedly repeated thanks to the Loop script. The Cameras and Actors text sprites serve as containers for DART behaviours and are hidden beneath the 3D World sprite, which covers the stage. The behaviours associated with each sprite are listed in the inset boxes, along with some of its most important inputs. In contrast to global monitoring carried out using a physical head- or camera-tracker, behaviours marked with an asterisk at the beginning of the line are only necessary when using marker tracking.

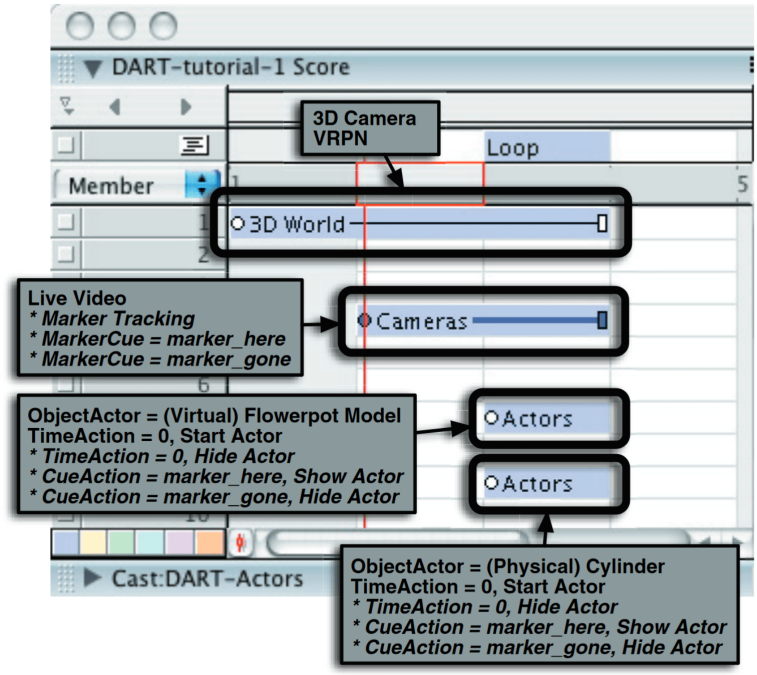


Figure 2.4: DART Score UI [13]

For DART Actor objects, DART offers an AR-specific event system. Cues (named events that are broadcast when a certain condition occurs) and actions (activities that take place in response to a named cue event or a fundamental condition) are two categories of DART-Events behaviors. A 3D object can also be added by designers, whether it be a simple object (such as a cube or cylinder) or an imported Shockwave3D model. The fact that the items can represent "virtual" or "physical" content is an intriguing feature. A physical object is used to simulate well-known physical items, whereas a virtual object is displayed normally and shown in the scene. [13]

2.3.4 AMIRE Framework

The AMIRE framework developed by Grimm et al. [7] is a software engineering approach to creating AR experiences that use established methodologies such as component theory and object-oriented application frameworks. The framework includes a generic application skeleton that can be customized using specialized authoring tools to create a final application. The building blocks of the system are called components, which have a well-defined interface for reuse. The AMIRE framework is designed specifically for creating AR experiences on wearable headsets like HoloLens.

A hierarchical authoring process with various abstraction levels is envisioned, which is built upon the usage of MR gems, MR components, and the MR framework. After the production of MR gems, MR components are produced by authors, and they are

gathered in a component library. The MR framework for authoring and final applications is produced by other authors. [7]

Existing gems collections such as game programming gems and graphic gems are similar to MR Gems. A specific MR can be efficiently solved by a gem, which is represented by it. Gems are made available in accordance with a list of essential problems that MR applications must answer. For instance, the "work path animation" gem, which depicts the workflow of a specific machine at an oil refinery, might be used to demonstrate the painting methods of a well-known painting in a museum. The MR gems can be used to develop both MR components that are particular to an application and an MR framework that outlines how MR components can interact with one another and be incorporated into an application. [7]

Visual authoring tools that employ certain authoring metaphors and are tailored to the unique requirements of reusable mixed reality elements provide quick development times. As a result, innovative design techniques for developing MR content, like iterative and rapid prototyping, are made possible. This frees authors from having to worry about the technical details of MR technology so they can focus on user requirements. [7]

2.3.5 AR Room

Park [16] came up with a modular system engine called AR-Room that makes it possible to build AR spaces and has deployable software components for interaction handling, tracking, scenario management, and rendering that are broken down into four categories. The main goal of AR-Room is to facilitate the implementation of augmented reality applications for developers who may not possess in-depth knowledge of underlying AR technologies. AR-Room comprises functional components, hardware abstraction modules, and an authoring toolkit, with its overall architecture depicted in Figure 2.5.

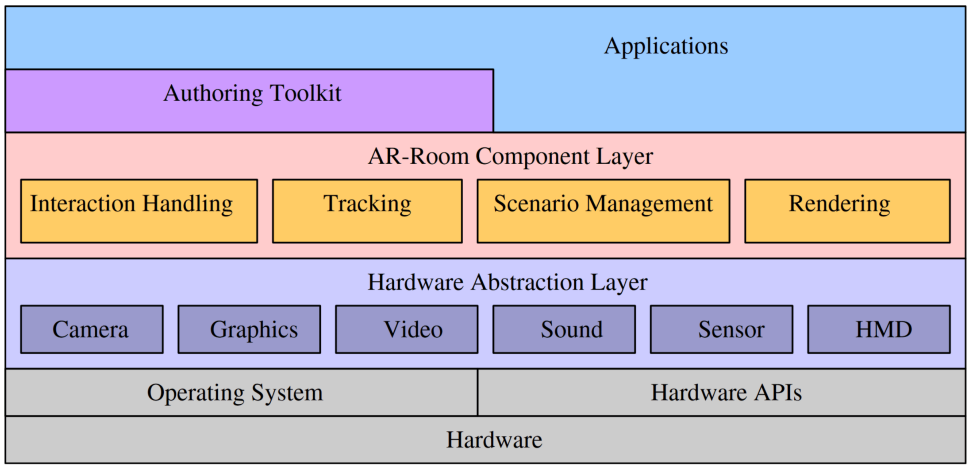


Figure 2.5: AR Room Architecture [16]

The AR-Room framework makes it simple to develop an AR application because the bulk of the core routines needed for an AR application has already been built and deployed as components. Putting together the proper AR-Room components and coming up with application-specific scenarios is thought of as the process of establishing a new augmented reality application. [16]

The creation of an augmented space involves the incorporation and arrangement of virtual objects within a physical space. Some physical objects can be replaced with virtual counterparts. A collection of objects constitutes a scene within the augmented space. In our scene description scheme, we distinguish between actual and virtual items. Each object is categorized as static or virtual based on how it interacts with the environment. Events can be produced by user activities, and virtual objects can react to those events. Physical things in the real space can also respond to events if the structure of the real space is known. Using an event-based action model, scenarios within the augmented space are created in script form. The model represents a scenario as a set of event-action pairs. The associated action is started when the appropriate event takes place. [16]

Virtual objects connected to pairs of events and actions make up a scenario. The action could, among other things, involve the playback of a video or music clip, the rigid-body modification of an object, a special effect, the switching to another virtual environment or scenario, or the alteration of some virtual objects' attributes. Once a scenario has been created into a script file, the scenario manager loads and manages the scenario while the script is operating. [16]

Several traditional engines, including a graphics engine, sound engine, video engine, logic engine, and physics engine, are used to create and simulate AR space. Fundamental operations are performed by these engines exactly as they would be in a regular 3D game. A virtual scene is built using the given scenario, and it is then blended with the actual image frame to generate the AR space. [16]

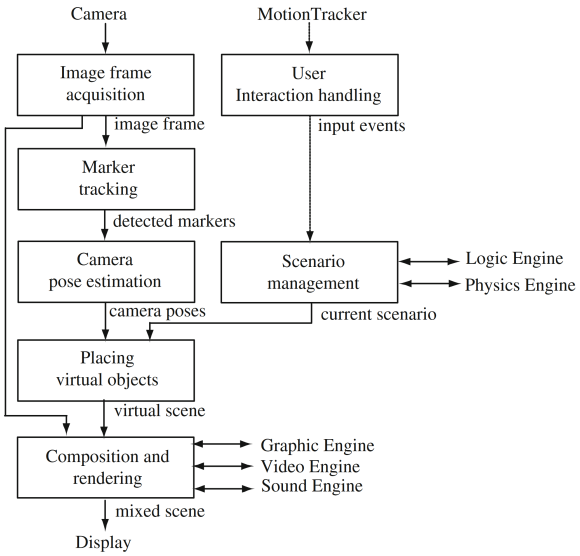


Figure 2.6: Steps of an application in AR Room [16]

The simulation process in AR-Room consists of the following steps: To begin, the virtual objects, encompassing environments, static and dynamic elements, are initialized. The process then proceeds with a recursive operation iteratively until the scenario reaches its conclusion. During this phase, user interactions are examined, collisions are detected, and input devices are monitored. Subsequently, events are invoked in response to the interactions and collisions that occurred. Following that, the properties of the virtual objects are modified based on the triggered events. Finally, the virtual objects are rendered, taking into account the updated properties, thereby completing the overall process.

Nevertheless, AR Room is quite broad in its scope, encompassing components such as hand tracking and others, and is not specifically tailored toward educational purposes. Additionally, the framework employs head-mounted displays.

2.3.6 ComposAR

ComposAR, developed by Seichter, Looser, and Billinghurst [20], is a general framework for AR applications and is not specifically targeted toward educational use. However, it has proven to be a versatile base tool for various activities such as educational, design-oriented, and research applications. Although the paper did not provide technical details on how ComposAR was developed or how it is used in detail, it is clear that the framework offers a range of features and functionalities, including the ability to associate virtual content with real objects and define interactions for them. ComposAR also offers a graphic user interface that allows for easy compilation and viewing of code and supports additional functionality through Python-based interaction or application plugins. Overall, ComposAR is a well-rounded framework that can serve as a strong foundation for a variety of AR applications.

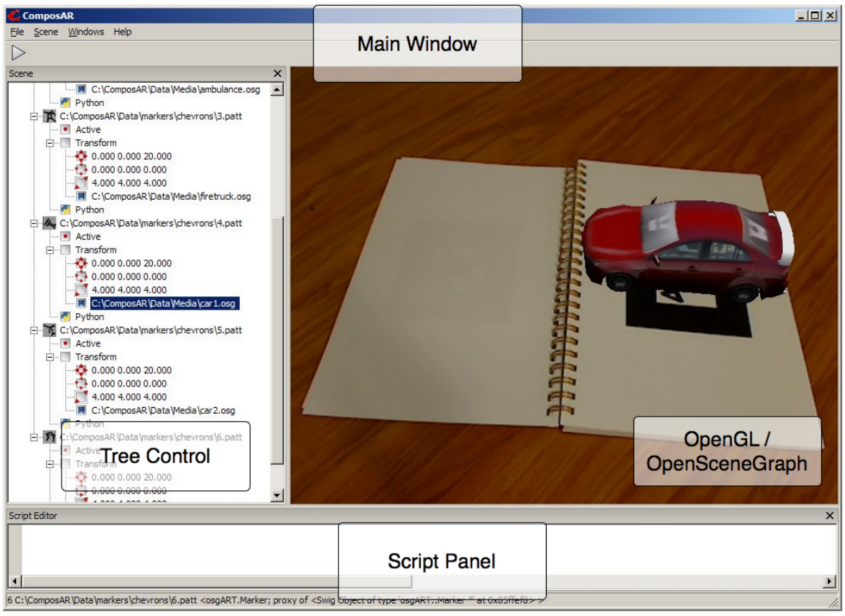


Figure 2.7: ComposAR Main UI [20]

2.3.7 FI-AR Learning

Coma-Tatay et al. [5] created FI-AR which is an Augmented Reality platform based on FIWARE that is specifically created for developing web-based AR educational content that can be accessed through Learning Management System (LMS). Unlike previous systems, FI-AR provides ordinary web interfaces without the need for plugins that provide access to the learning framework. Additionally, it has an integrated evaluation platform using LMS and an infrastructure for identification, billing, and revenue-sharing through the FIWARE business ecosystem, which is not available on LMS. The goal of FI-AR is to make it easier to develop AR-based classes that can be included in current LMS platforms like Moodle.

A cloud-based architecture called FIWARE enables the development and distribution of Internet services and applications. The FIWARE community seeks to create a sustainable, open environment based on open-source software standards. [5]

The content editor, administration system, store interface, and content player make up the bulk of the FI-AR Learning framework. The controlling system manages the publication of exercises, while the content editor is a tool used to create augmented reality exercises. The store interface grants access to the distribution of content. Conversely, the content player serves the purpose of executing the exercises crafted with the editor, providing learners with a means to engage in the practice. Furthermore, an external and autonomous application known as LMS is employed to play and display the exercises using the content player. [5]

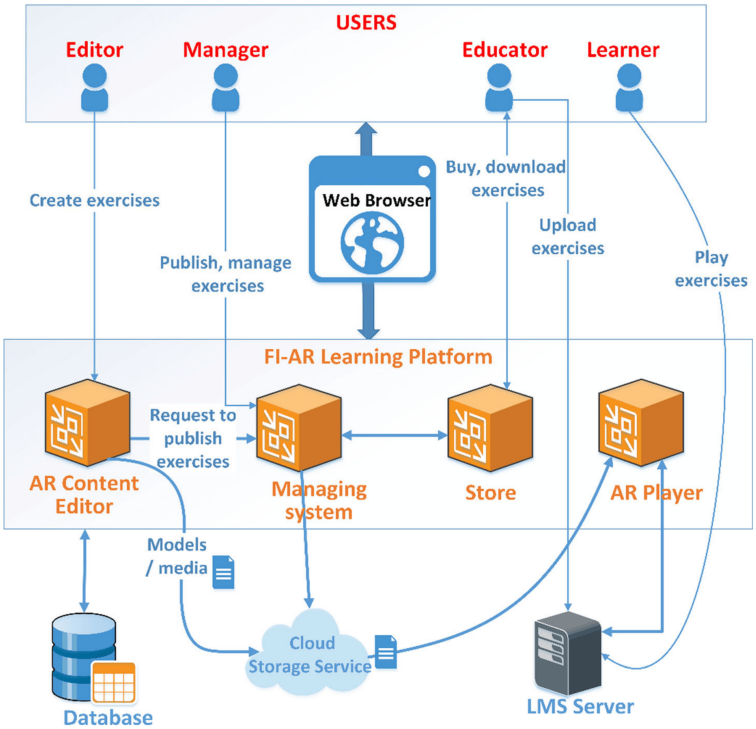


Figure 2.8: FI-AR Architecture [5]

The FI-AR Learning framework's workflow entails several processes. Using the AR Content Editor, an editor first produces educational content. A manager then uses the managing system to publish the content. Teachers buy the materials from the store, download them, and then upload them to their LMS platform. Finally, students use their LMS platform to access the exercises, which are then shown in a web browser using the AR Player. The system architecture is illustrated in Figure 2.8. [5]

Yes/no questions, questions with multiple answers, and interactive point-at questions are all supported by FI-AR, which is primarily aimed at developing exercises. In the latter, in order to offer the proper response, The student needs to identify and point at an object (or a particular component of an object) in a 3D scene. The concept is to associate the AR content with the corresponding question, as shown in Figure 2.9. [5]

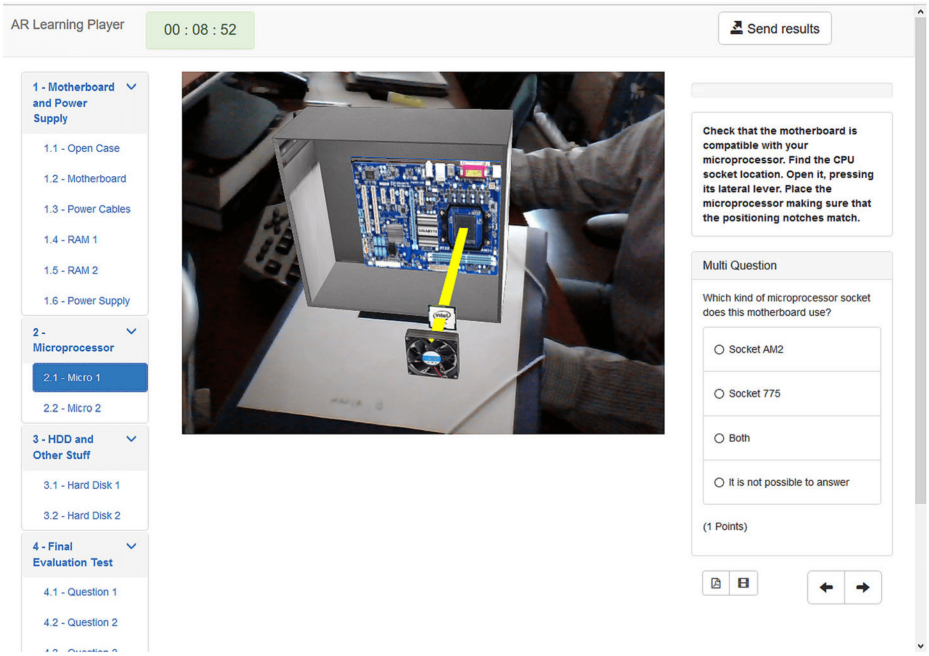


Figure 2.9: FI-AR Content Player [5]

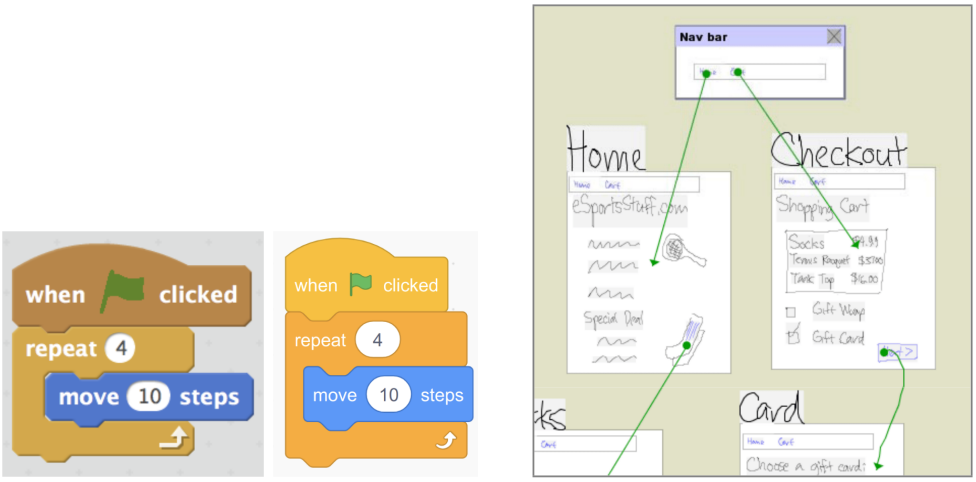
Although FI-AR is very user-friendly and does not require programming knowledge, its capabilities are limited to a specific set of exercises. Nonetheless, what sets FI-AR apart is its integration with an LMS, as well as its store that facilitates cooperation among users and enables a good feedback loop.

2.3.8 Visual Programming Languages

Visual Programming Language (VPL) is a specific type of programming language that grants users the ability to create programs primarily through the manipulation of graphical elements [23]. Common interaction models used in VPLs include dragging blocks on

a screen (as in Scratch), utilizing flow diagrams, state diagrams, and component wiring (as in Pure Data), and using icons or other non-textual representations (as in Kodu [9]). Although many VPLs still incorporate text or a combination of text and visual representations, each VPL consists of grammar and vocabulary that together determine the set of ideas that can be expressed with the language. Grammar is the visual metaphor employed by the language, such as blocks or wires, while vocabulary comprises the set of components, icons, or blocks that permit the expression of concepts. [17]

Blockly is a widely used visual programming language that allows users to create programs through graphical manipulation. The language is based on a block-like interface, which users can drag and drop to create their code. Blockly is designed to be easy to use and is particularly well-suited for teaching programming concepts to beginners. The language has a rich set of features, including support for loops, conditionals, and variables. Additionally, it allows for the addition of unique blocks and extensions, making it quite customizable. Blockly has been used for a variety of purposes, including teaching coding in schools and creating sophisticated software systems. As a result, it has grown to be one of the most popular visual programming languages, with a huge and active user base. [4]



(a) Blockly VPL (b) Denim VPL

Figure 2.10: Visual Programming Languages

DENIM is a web-based application that enables designers to generate web page prototypes by roughing out and fine-tuning their design concepts using a collection of straightforward drawing primitives. The tool provides a set of widgets, such as text boxes, buttons, and images, that can be used to create interactive prototypes. The widgets can be linked together to create a navigation structure for the prototype, and the designer can annotate the prototype with notes and comments. DENIM allows designers to create prototypes quickly and easily, without the need for specialized software or programming knowledge. The tool has been used in a variety of design contexts, including website

design, mobile application design, and game design [15]. Lin, Thomsen, and Landay [11] leveraged DENIM as the foundation for their modification that introduces novel components enabling designers to efficiently reuse storyboard fragments, improved arrows with a broader range of event types, and conditionals to facilitate page transitions that are contingent on the state of interface elements in a page.

2.4 Summary

This chapter introduced Augmented Reality and its impact on various fields, particularly Education. The discussion specifically focused on the influence of AR on Education. Additionally, the chapter presented Unity as a tool for building AR applications. The Oculus Quest, a device capable of running AR applications, was also introduced.

In the previous section, various frameworks were discussed for creating augmented reality (AR) content. The Meta AR [22] and AuthorAR [12] frameworks were developed with the education sector in mind. However, MetaAR had limitations as it could only handle animation actions and a limited set of tasks, whereas AuthorAR was focused on special needs and offered only a few tasks such as associating text with images.

DART [13] and ComposAR [20] were promising frameworks as they provided robust tools for creating generic AR experiences. However, they failed to provide adequate support for education and their interactions were limited to basic functions. Additionally, both frameworks lacked a visual programming language that would be accessible to non-programmers.

The AMIRE framework [7] provided a component-based approach that showed promise, but it too was geared toward creating generic AR content and did not offer the necessary tools for educational purposes.

The FI-AR framework [5] was unique in that it provided a learning management system that enabled users to browse and download AR experiences. Teachers could also upload their own creations. However, the framework was limited in terms of interaction potential, as it only allowed students to be asked questions while viewing an AR scene.

Overall, no framework was identified that offered a visual programming language capable of handling a decent amount of complexity, provided adequate support for AR, included components for educational purposes, and offered online support for distributing AR experiences.

Chapter 3

System Design and Implementation

This chapter discusses the system design goals, architecture, implementation decisions, and technologies used to execute the system design.

3.1 Design Goals

The objective is to create a platform that enables teachers to design and showcase augmented reality (AR) experiences featuring science experiments. The project comprises two primary components: a designer built for desktop and an AR player built for the Oculus headset. The platform should have the ability to produce general AR experiences while also providing features essential for crafting science experiments, such as a physics engine.

The final product should empower teachers to generate experiments utilizing their customized models and logic, specified through visual programming. Subsequently, students can download and play these experiments on their personal Oculus headsets or those owned by their educational institution.

3.2 System Overview

The overall system is straightforward. The project is divided into two components, the player and the designer. The player is implemented as an Android game in Unity that runs on the Oculus Quest Headset. On the other hand, the designer is created as a React web app but it also contains a Unity app running on WebGL for preview. Communication between both is done via a cloud service such as Firebase [6]. The overall system architecture is shown in Figure 3.1.

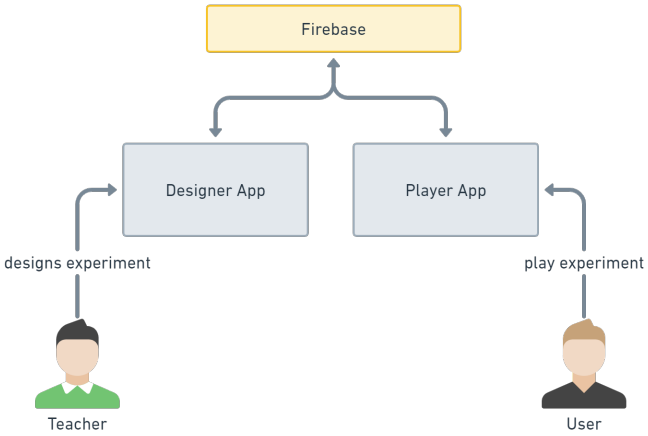


Figure 3.1: System Overview

3.2.1 Designer Workflow

Once an experiment is selected for design, The designer application has three primary interfaces: the object types list interface, the scene manager interface, and the scene interface.

Object Types List Interface

The teacher can create and manage object types in the object types list interface. An object type is simply a 3D model. These 3D models are later used to create objects placed in the scene. The teacher can create a new object type by uploading a 3D model with the .glb file format. The object types list interface is shown in Figure 3.2.

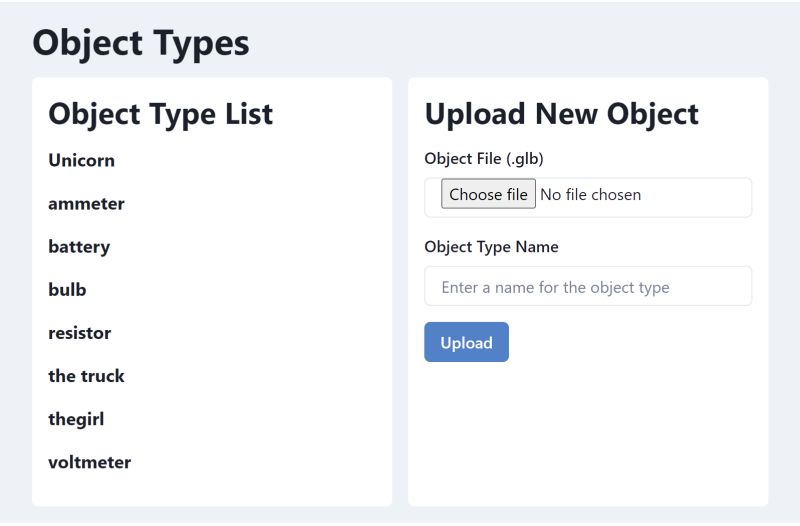


Figure 3.2: Object Types List

The .glb file extension was used for 3D models because it is more efficient than other file formats such as .obj and .fbx. Unlike other file formats, the glb file format is self-contained, which includes all the necessary data for a 3D model, such as the geometry and textures within a single file. This format is commonly used on the web and in virtual or augmented reality.

Scene Manager Interface

Every experiment contains a set of scenes. Every scene has a list of objects that are shown. The scene manager interface lists the scenes in the experiment and allows the teacher to create new scenes. The teacher can also open a scene by clicking on the button with the magnifying glass icon.

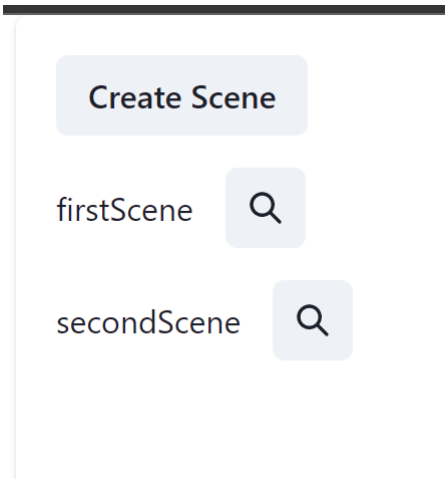


Figure 3.3: Scene Manager Interface

Scene Interface

Once a scene is selected and opened. Multiple components will be visible in the application.

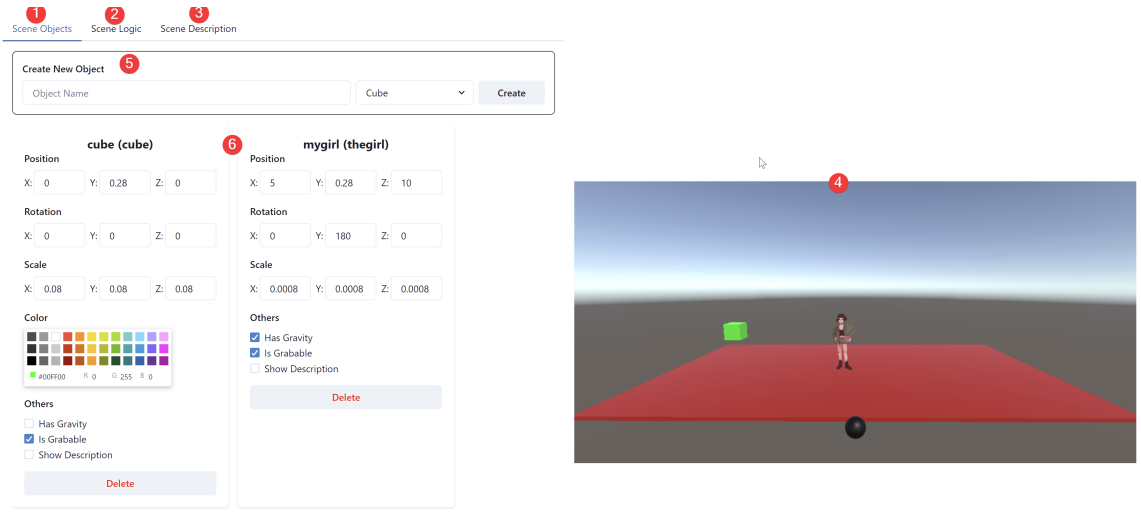


Figure 3.4: Scene Window

The first component is the Scene Objects interface which is the first visible interface. There is where the teacher can create new scene objects via the interface (5) in Figure 3.4. Already created objects can be modified using the list at (6) in Figure 3.4. A set of properties such as the position, rotation, scale, or if the object should be affected by gravity can be changed for each object.

The second interface is the Scene Logic interface accessed by the tab numbered (2) in Figure 3.4. This is where the scene's logic is defined using a node-based visual programming language. New nodes can be created from the context menu, and connections can be easily created using the sockets in the nodes. More details on the visual programming language will be discussed later.

The Scene Description interface that is accessible via (3) in Figure 3.4 is where the teacher can set a description for the experiment. This description will be shown to the user once the scene is loaded.

Finally, a scene preview is found at (4) in Figure 3.4. This window is a Unity app running in WebGL mode. It is kept in sync with the objects and properties currently created. This helps the teacher see a preview of the scene without loading the player.

Logic Designer

The logic designer is the essential part of the designer as this is where the teacher creates the scene's logic. The logic is set via a visual programming language consisting of nodes to do certain actions or get certain properties. Nodes are connected together to define the behavior. An example of scene logic can be seen in Figure 3.5.

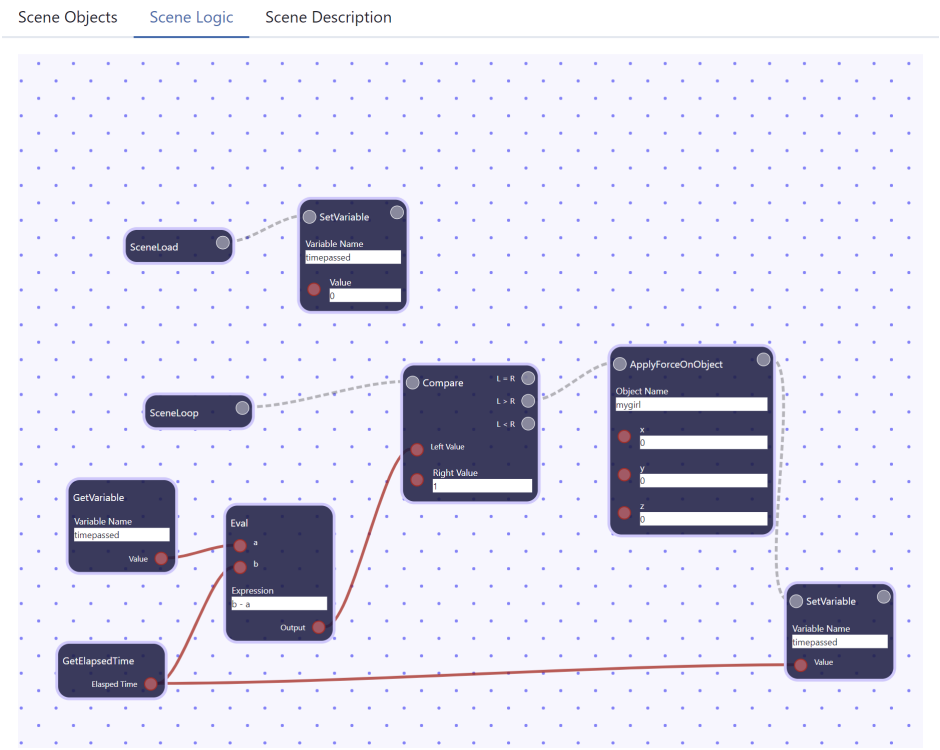


Figure 3.5: Scene Logic Designer

The nodes are split into two types, execution nodes, and data nodes. Execution nodes are the ones responsible for determining the flow of the logic. These nodes will have grey sockets at the top, and connecting them will create a grey connection, as seen in Figure 3.5 between the SceneLoad node and the SetVariable node. Data nodes are responsible for simply returning certain properties that execution nodes could require. This is useful because it enables us to have generic nodes that create complex logic when connected.

Every node can take inputs; the inputs always have sockets beside them, indicating their values can come from other nodes. Some nodes also have inputs that do not have a socket. These inputs are called controls.

All execution begins from SceneLoad and SceneLoop nodes. SceneLoad executes all its children at the start of the scene. SceneLoop calls all of its children every fixed number of milliseconds.

The designer has various nodes to ensure a wide range of experiments can be designed. An example of the nodes includes the following:

- **GetPosition**: A data node that returns the position of the object with the name specified.
- **GetScale**: A data node that returns the scale of the object with the name specified.

- **SetPosition**: An execution node that sets the position of the object with the name specified. The coordinates are passed in from a data node or entered explicitly.
- **SetStaticFriction**: An execution node that sets the value of the static friction of the object with the name specified. The value of the friction can be specified explicitly or retrieved from a data node.
- **Compare**: An execution node that takes two values as input from any data node and compares them. Then depending on the result, one of three paths is taken: equals, bigger than, or less than.
- **Eval**: A data node that takes in two values as inputs from other data nodes and a specified math expression. Then, the value is provided to be used by other nodes.
- **(Set/Get)Variable**: A pair of data and execution nodes used to manage custom variables in the scene. Useful to define custom properties.

3.2.2 Player Workflow

The player workflow is simple. The user has a menu to select the experiment to play, as seen in Figure 3.6.

Figure 3.6: Select Experiment Window

After selecting an experiment, a description set in the designer is shown first. The experiment is visible once that window is closed. The user can interact with the experiment freely, and the logic set in the designer applies. See section 3.2.1 for details.

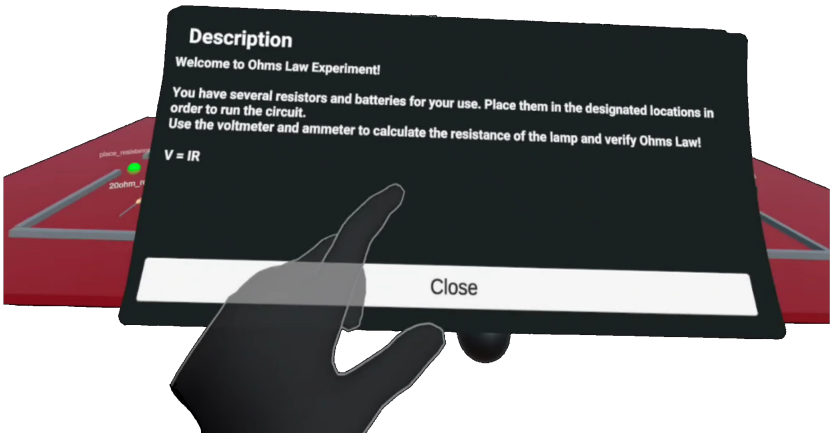


Figure 3.7: Experiment Scene

3.3 Database Design

3.3.1 3D Models

The 3D models uploaded by the teacher are stored in Firebase storage.

> Firebase is a platform for app development known as Backend-as-a-Service (BaaS), which offers hosted backend services including a real-time database and cloud storage. [6].

Every file is uniquely identified by its name. All files end with the extension .glb as that is the supported format by the application.

3.3.2 Experiment Data

The scene data consisting of objects, their properties, and the scene logic are all stored in a Firestore instance of Firebase.

The data is stored in a collection called experiments. Each experiment is a document in the collection. The document contains the following fields:

- **name**: The name of the experiment.
- **scenes**: A list of scenes in the experiment. Each scene is a document in a collection called scenes.

Each scene in the scenes collection consists of the following fields:

- **name**: The name of the scene.
- **description**: The scene description shown when the scene is first loaded.
- **index**: The order of the scene in the scene list. The scene with the lowest index is the one that is loaded first.
- **sceneLogic**: An object containing all details of the scene logic.
- **objects**: A collection of objects in the scene. Each object has its name as the unique identifier, and the content follows the following format:
  - **objectName**: The object's name in the scene.
  - **objectType**: The name of the 3D model this object should load. It can also be one of the following primitive types: cube, sphere, cylinder, or capsule.
  - **color**: The initial color of the object in hexadecimal.

  - **hasGravity**: A boolean value that represents whether the object is affected by gravity.
  - **isGrabable**: A boolean value that represents whether the user can move this object with hands.
  - **showDesc**: Whether to show the object's name above it.
  - **position**: The initial position of the object.
  - **rotation**: The initial rotation of the object.
  - **scale**: The initial scale of the object.

Scene Logic

The scene logic is stored in the scene data. The 'sceneLogic' property is a flattened map containing all the nodes of the visual programming language. The connections are also defined in the structure. A random alphanumeric string identifies every logic node. For every logic node, the following is stored:

- **name**: The type of the node. Review example types in section 3.2.1
- **position**: The node's position in the graph-based visual programming area. This position is required so that all nodes stay in the same location. An example of the logic area is found in Figure 3.5

Every node also contains a couple of maps. A map is a key-value pair data structure.

- **controls**:
  - Key represents the name of the control.
  - Value is the value for this control
- **execOutputs**:
  - Key represents the name of the execution output socket.
  - Value represents the identifier of the target node to be called.
- **inputValues**: This map is used when the input is not connected to another node.
  - Key represents the name of the input.
  - Value is the value for this input.
- **inputsFrom**: This map is used when the input is connected to another node.
  - Key represents the name of the input.
  - Value is another object of the format:
    * nodeId: The node's identifier where the input will come from.
    * outputName: The name of the output at the node with the identifier 'nodeId' to get the data from.

3.4 Designer Implementation

The designer is implemented as a web application using React and Typescript. A few hooks were created to manage different parts of the designer app. An overview of the system design of the designer is seen in Figure 3.8. The subsequent subsections provide details about each module.

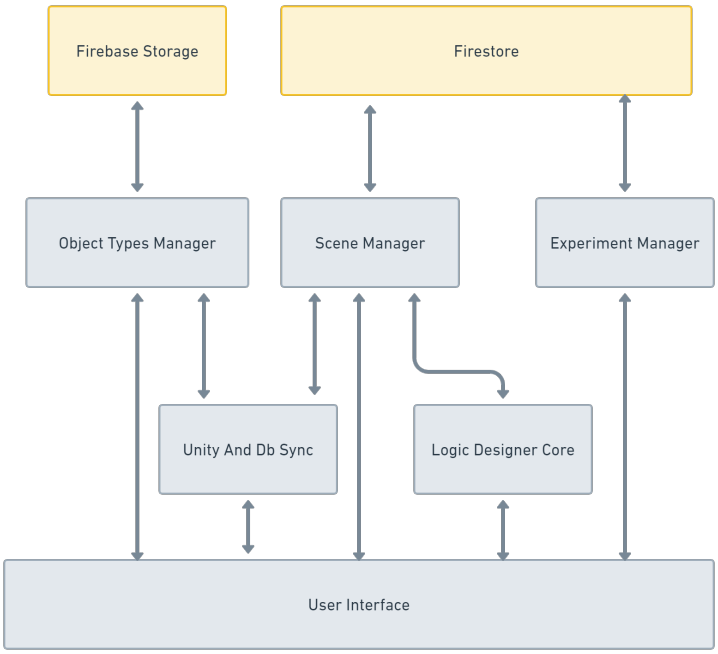


Figure 3.8: Designer System Design

3.4.1 Object Types Manager

This module is responsible for managing the 3D models of the application. Every teacher can upload new 3D models into the system, and then other teachers can use them in the scenes of an experiment.

The module is written as a React Hook that is used in two places: the interface where teachers can upload 3D models and the component responsible for syncing the Unity instance and the scene data.

This hook connects to Firebase Storage to create and retrieve the 3D models. The following interface is returned:

```
interface ObjectTypesManager = {
  objects: ObjectType[];
  uploadObject: (objName: string, objFile: Blob) => Promise<boolean>;
};
```

3.4.2 Scene Manager

Scene data is managed through this React Hook. The UI can access and modify the scene data through functions available via the hook.

The Scene Manager is also connected to the Logic Designer as the scene logic is saved into Firestore.

In addition, object data is exposed via this hook. The UI has the ability to modify the object's initial properties as well as its name and type.

The hook returns the following interface:

```
interface SceneManager = {
    scene: SceneState,
    objects: SceneObjectState[],
    setDescription: (description: string) => void,
    addObject: (name: string, type: string) => void,
    setSceneLogic: (nodes: ExportedNodes) => void,
    getObject: (objectName: string) => SceneObjectInterface,
};

interface SceneObjectInterface {
  object: SceneObjectState | undefined;
  setPosition: (position: [number, number, number]) => void;
  setRotation: (rotation: [number, number, number]) => void;
  setScale: (scale: [number, number, number]) => void;
  setHasGravity: (hasGravity: boolean) => void;
  setGrabbable: (isGrabbable: boolean) => void;
  setColor: (color: string) => void;
  deleteSelf: () => void;
  setShowDesc: (showDesc: boolean) => void;
}
```

Every function of both interfaces sends the data over to Firestore. Then other modules listen to data from Firestore.

3.4.3 Unity and Db Sync Manager

This is another React Hook responsible for keeping the unity instance in sync with the data in Firestore. It listens to changes in Firestore and then replicates the changes in the Unity app. Communication is established between the two via javascript.

A `useEffect` hook exists that continuously listens to changes in the data from the scene manager component. Then once there is a change, the data is forwarded into another class called Unity Object Management which exposes functions that directly interact with the Unity instance. The forwarded functions are `addObject`, `deleteObject`, `setPosition`, `setScale`, `setRotation`, and a few others covering the whole of `SceneObjectInterface`.

Adding an object

Objects need 3D models to be created. Thus, when an object is created, this component has extra logic for handling the 3D model info. The Unity instance in the Designer App does not have access to Firestore, so the event must communicate with Firebase Storage to get the download link of the 3D model. Then the link is passed to the Unity App for downloading. All this happens before the actual `createObject` in Unity is called.

Moreover, the preview app implements a caching mechanism to prevent redundant downloads of the same 3D model. Once downloaded, the 3D models are stored in the indexed database of the browser. Consequently, the cache is initially checked when the Unity app receives a request to download a 3D model. The 3D models are indexed based on their names (the name of the 3D model being the type within the addObject function).

3.4.4 Logic Designer

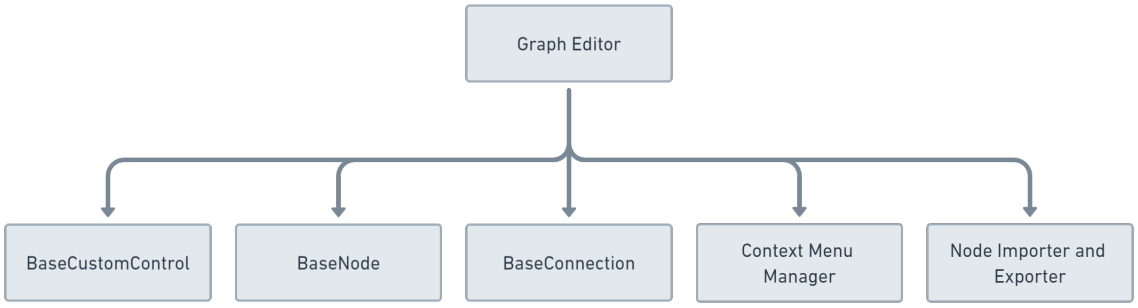


Figure 3.9: Logic designer components

The logic designer is built on top of a library called Rete.js. This library provides a base framework for creating graph-based visual programming languages. [19]

A design was created for every node where every node inherits the BaseNode component. The node's look can be seen in Figure 3.5. Every node has inputs, outputs, and controls. Inputs are sockets in the nodes that can be connected to outputs of other nodes. Controls are also inputs but are set to fixed values by the Teacher. Inputs can also be associated with controls of their own, enabling the Teacher to either get the input from another node or set it explicitly. In addition, some nodes have special grey-colored sockets indicating the execution path.

Connections were created with a distinct design to differentiate between data connections, which move data from outputs to inputs, and execution connections, which move execution from one node to another. Data connections are shown in red, while execution connections are grey-colored and are animated to show the direction of execution.

The language also features a context menu that enables the Teacher to create any nodes required for their logic. The menu is categorized to make it easier to find the required nodes. An example of the nodes available can be seen in section 3.2.1

Node Importer and Exporter

This module is responsible for converting the graph structure we have in the visual programming language to the flattened structure we described in section 3.3.2

A map data structure is created first; then, for every node, a corresponding object is created in the map. Then, every input and control of that node is filled if they are set to a fixed value. The fields filled are `inputValues` and `controls`.

After this, a loop is made on connections to fill in the inputs that take their input from other nodes and execution connections, which are `inputsFrom` and `execOutputs`, respectively.

Finally, this object is returned and saved directly to Firestore. A similar process but reversed occurs when loading the data back from Firebase.

3.5 Player Implementation

The player is created in Unity as an AR/VR application using the Oculus API. The experiment is loaded into a red platform similar to the scene preview window in the designer then the logic is also loaded and executed.

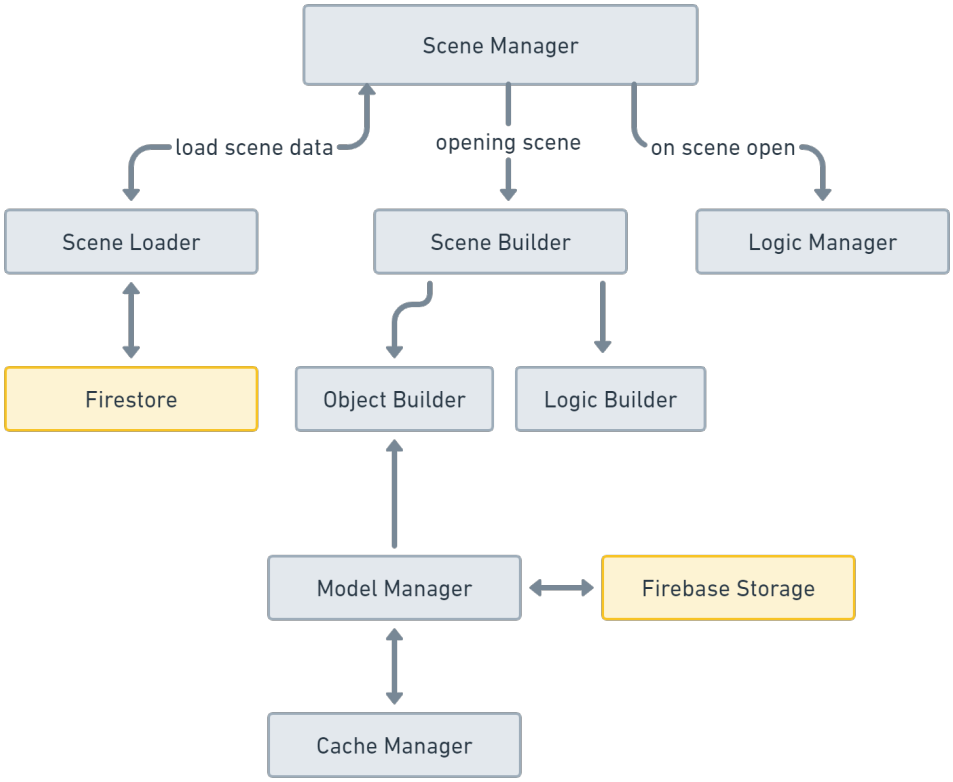


Figure 3.10: Player system overview

The Unity app has a main component to run the scenes. The scene manager is responsible for loading and moving in between scenes. It depends on the logic manager which is responsible for the scene's logic after it is created, as it runs all the SceneLoop flows in every frame. A complete overview of the system design can be seen in Figure 3.10

3.5.1 Scene Manager

The scene manager module is the main driver of the player application. The module is responsible for loading the scene data, then building it into an executable format. It is also responsible for running the logic via the Logic Manager.

Class Functions

1. `Start()`

   Main method which starts everything. It first loads the data of the current experiment and then proceeds to execute the first scene with the lowest index.

2. `SetCurrentScene(SceneData sceneData)`

   This method is used to destroy the currently loaded scene and its objects and then creates the scene passed to it. The `SceneData` type is the data of the scene returned directly from Firebase. It follows the exact format in subsection 3.3.2

3. `StartSceneExecution()`

   Starts the scene execution by calling into the Logic Manager.

4. `ShowDescription(string message)`

   Shows a message for the user in the Oculus headset. The message is shown to the user in the same description window shown in Figure 3.7

5. `HideDescription()`

   This function is called when the Close button of the description panel is clicked.

3.5.2 Scene Loader

This module is responsible for the loading of the scene data from Firebase as well as providing access to them for the other modules.

Class Functions

1. `LoadAllScenes()`

   This function loads all the scene data from Firestore into a list of SceneData. However, the scene objects are not loaded in this function. Instead, for every loaded scene, `LoadScene` is called in order to load its objects.

2. `LoadScene(SceneData scene)`

   Loads the objects of the scene one by one and stores them into the passed SceneData object. Thus, the scene loader also acts as the object loader.

3. `GetSceneIndex(int idx)`

   Returns the SceneData of the scene at the index `idx`.

4. `GetSceneWithName(string nme)`

   Returns the SceneData of the scene having the name `nme`.

3.5.3 Scene Builder

The scene builder exposes one function that is responsible for building SceneData into an actual scene. First, a `Scene` object is created to hold all the scene data except the objects. Then, the object builder is called for every object data in the scene data in order to fill the objects list in the `Scene` object. The `SceneBuilder` also calls the `LogicBuild er` in order to load the logic data properly.

Class Functions

1. `CreateSceneFromData(SceneData sceneData)`

   Takes in the scene data and then creates a Scene based on the data.

3.5.4 Object Builder

This module also exposes one function that takes in an ObjectData and returns a created object. The module creates the object in Unity itself as well as stores any required references.

First, the base GameObject is created. A primitive is used if the type is a primitive object, otherwise, the model manager is used to load the GLB 3D model. Then, a MeshCollider and a Rigidbody are added to the GameObject in order to add collision and gravity using Unity's engine.

In addition, a separate GameObject is created to handle the description shown on top of the objects. The description's position and rotation are kept in sync using a component called `LabelSyncComponent`.

Then, the created data is encapsulated inside a class called `Object` which exposes several functions to change the properties. The object builder calls these functions with the initial values specified in the designer.

Class Functions

1. `CreateObjectFromData(ObjectData objectData)`

   Takes in the object data and then creates the Object before returning it.

The Object

The `Object` class is responsible for providing API that interacts with a specific scene object in Unity. The functions provided are as follows:

1. `UpdatePosition(List<float> position)`

   Takes in a list of three floats and force sets the position of the object. The values of the position are passed through a class called `PositionConverter` first in order to convert the values from the designer format to actual Unity coordinates.

2. `GetPosition()`

   Returns the current object position after being converted through `PositionConverter`.

3. `GetRealPosition()`

   Returns the actual position of the object in the Unity Worldspace without any conversions to the designer system. This is useful when calculations are required to be done on the positions.

4. `UpdateScale(List<float> scale)`

   Updates the scale of the object to the values passed in the list.

5. `GetScale()`

   Returns the Vector3D of the scale of the object.

6. `UpdateRotation(List<float> rotation)`

   Updates the rotation of the object to the three values of the List.

7. `GetRotation()`

   Returns the current rotation of the object as a Vector3D.

8. `UpdateColor(string color)`

   This function takes in a color as a hex code and then sets the color of the object if it is a primitive.

9. `UpdateVisible(bool state)`

   If the state parameter is set to false, the object is hidden from view using the `Set Active` function in Unity. Otherwise, it is shown again.

10. `UpdateStaticFriction(float value)`

    A function used to set the static friction of the object which is a constant needed in physics simulation in Unity.

11. `UpdateDynamicFriction(float value)`

    Similar to `UpdateStaticFriction` but changes the dynamic friction.

12. `UpdateBounciness(float value)`

    This is also a physics property that controls how bouncy an object is when it collides with another object with a force.

13. `UpdateMass(float value)`

    Updates the mass of the object which is used in Unity's physics simulation.

14. `GetSpeed()`

    Returns the current object speed magnitude.

15. `SetDescription(string desc)`

    Updates the object label that is shown on top of the object and moves with it.

16. `ApplyForce(Vector3 force)`

    Applies the force passed to it on the object. The force is specified in Newtons as documented by the Unity engine.

17. `UpdateGrabable(bool isGrabable)`

    This function enables or disables the grabbing feature in Oculus. When enabled, the following components are added to the GameObject in order to make it moveable using bare hands inside the Oculus.

- `Grabbable`: Adds the necessary code to the object to make it Grabbable in the Oculus.
- `PhysicsGrabbable`: Enables physics when the object is grabbed. This helps make the objects throwable.
- `HandGrabInteractable`: Marks the object that it must be grabbed by hands and not the controllers.

18. `Destroy()`

    Destroys everything related to the object and the object itself.

3.5.5 Model Manager

This module is responsible for loading the models from Firebase Storage. It also works with the `CacheManager` in order to check if the 3D model is already cached before downloading it from Firebase.

Class Functions

1. `GetModelBytes(string modelName)`

   The only function in the class. It is responsible for getting the 3D model as an array of bytes. The cache is checked first and if a miss occurs, the model is downloaded from Firebase Storage. If the model is downloaded, it is cached in order to be used later.

3.5.6 Cache Manager

A simple module that gives access to the Oculus's cache folder.

Class Functions

1. `GetFileFromCache(string fileName)`

   Returns the file specified from the cache as an array of bytes. Or it returns `null` if the file does not exist.

2. `PutFileInCache(string fileName, byte[] file)`

   Stores in the cache, the array of bytes specified with the file name specified.

3.5.7 Logic Builder

The logic builder is responsible for converting the flattened structure described in section 3.3.2 into a graph structure with certain starting nodes forming a Chain of Responsibility design pattern. Starting nodes are always either SceneLoad or SceneLoop with the first being called once and the latter being called every frame. Nodes are called Instructions on the side of the player.

The Instructions

An abstract class called `DataInstruction` was created to be the base of all instructions. It defines the basic properties and methods required in each instruction such as inputs, and controls. The outputs are made available from a `DataInstruction` by overriding the function `GetOutput` which is called from the dependent instructions. Thus, in essence, data instructions are **executed** every time they are asked for their output (Figure 3.11).

In addition, data instructions have inputs that could come from other data instructions. These inputs are wrapped in a class called `InputParam`. The `InputParam` class is a simple structure with the following constructor:

```
InputParam(DataInstruction instruction, string paramName)
```

It holds the `DataInstruction` that the input should come from as well as the output name *(paramName)* to get from that instruction. The class exposes a public method called `GetValue()` that simply calls `GetOutput(paramName)` on the given data instruction. This exposed method can then be used in the instruction's body to get the value of the inputs.

Some instructions like `SetMass`, `SetPosition`, and `ApplyForceOnObject` do not just return some property or data. They do some action that affects things in the scene. Thus, another abstract class called `ExecInstruction` was created and inherits from `DataInstruction` but also adds several new properties and functions to add execution to these nodes. This is done by defining a new method called `Execute` as well as an abstract method called `ExecuteImpl` (Figure 3.11). The `Execute` method simply calls `ExecuteImpl` and then calls the next instruction (`nextInstructs["exec"].Execute()`). Some instructions like the `CompareInstruction` *(See Figure 3.13)* define multiple execution paths and thus need to override the `Execute` method.

It's important to note that exec instructions also have a `GetOutput` function, similar to data instructions. As a result, an `ExecInstruction` has two methods of execution: the logic in the `ExecuteImpl` method and the `GetOutput` method. However, it is common that the `GetOutput` method of exec instructions simply returns the values that were computed during the last call to `ExecuteImpl`.

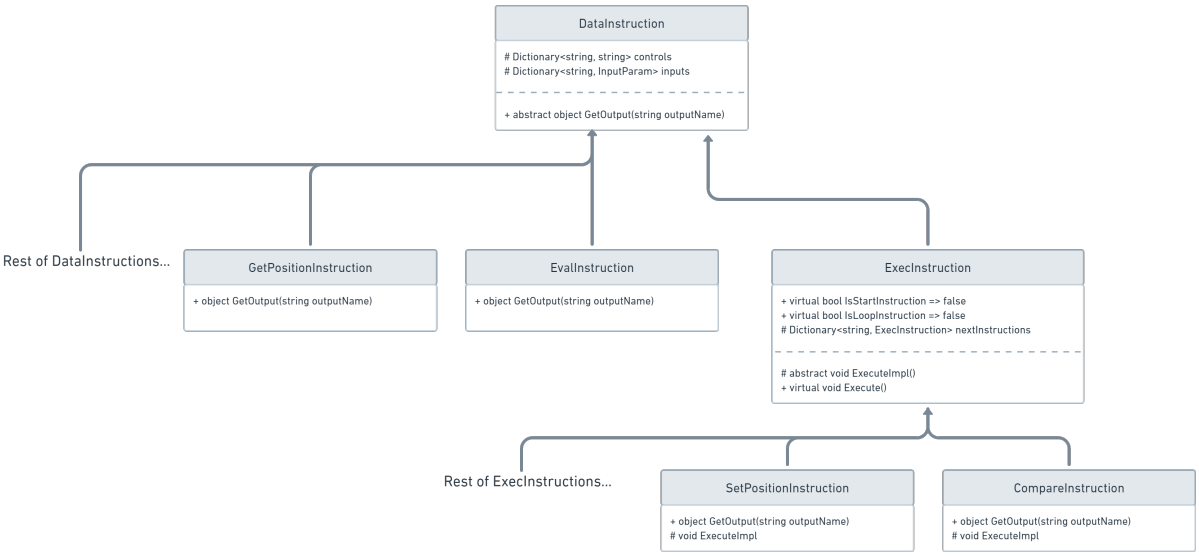


Figure 3.11: Instructions class diagram

All the functions listed in section 3.5.4 and more are implemented as instructions.

Logic Builder Functions

1. `GetInstructions()`

   The main function of the builder class. The class is given the flattened structure coming from Firestore and then this function converts them into the linked-graph structure consisting of Instructions as seen in Figure 3.13.

2. `CreateInstruction(string type, Dictionary<string, string> controls, Dictionary<string, InputParam> inputs, Dictionary<string, ExecInstruction> nextInstructs)`

   A factory function that creates one of the instructions that inherit `DataInstruction` or `ExecInstruction` depending on the parameter `type`.

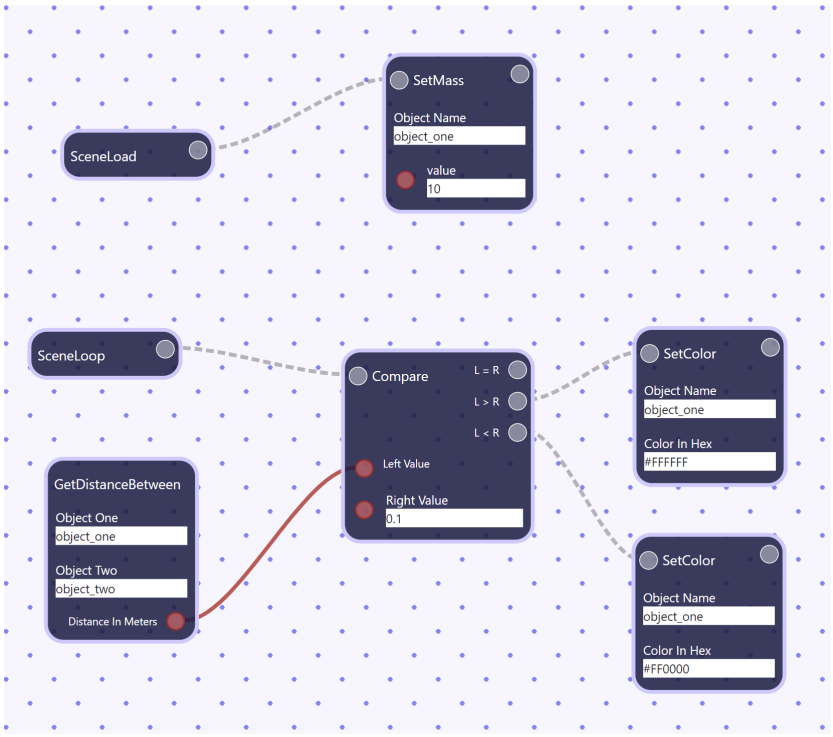
3. `DataInstruction ConvertInstruction(string id, SceneLogicData logicNode)`

   A recursive function that is used to convert a logic node retrieved from Firestore to one of the Instructions.

Building Implementation

Logic data, as said before, comes from Firestore in a flattened structure. The structure has an identifier for every Node *(now known as instruction)* and the connections are referenced by these ids. See section 3.3.2 for the exact format. Figure 3.12 also shows an

actual example of data brought from Firestore and the corresponding exported diagram in the designer. Some fields are omitted.



(a) The logic design in the designer app before export

(b) Flattend JSON

Figure 3.12: Example of flattened logic data coming from desinger

When the processing is finished, a composite-like structure is created forming a chain of responsibility design pattern. An example structure can be seen in Figure 3.13 which is the result of processing the data in Figure 3.12. All the `SceneLoad` instructions are executed at the beginning of the scene while the `SceneLoop` instructions are executed every frame. The instructions connected to either of these nodes are executed subsequently.

In Figure 3.13, it is noticed that a new instruction called `ConstantValueInstruction` exists. This instruction is used when the value of an input is fixed by the Teacher to a certain value instead of being connected to a `DataInstruction`. In this case, a `ConstantValueInstruction` will be created to simply simulate a `DataInstruction` returning the fixed value.

One may wonder about the difference between `controls` and `inputs`. It is that `controls` are always fixed to a value by the Teacher. Thus, they are stored as a simple map of strings. Meanwhile, `inputs` can either come from another instruction or also be fixed by the teacher. This is where `ConstantValueInstruction` comes to play as it

is used to encapsulate fixed inputs ensuring a consistent interface for all inputs. This guarantees that inputs always originate from other instructions. (An example of this is in the compare instruction in Figure 3.12 where the left value comes from the `GetDistanceBetweenInstruction` while the right value is fixed to 0.1 and must be encapsulated into a `ConstantValueInstruction` as explained above)

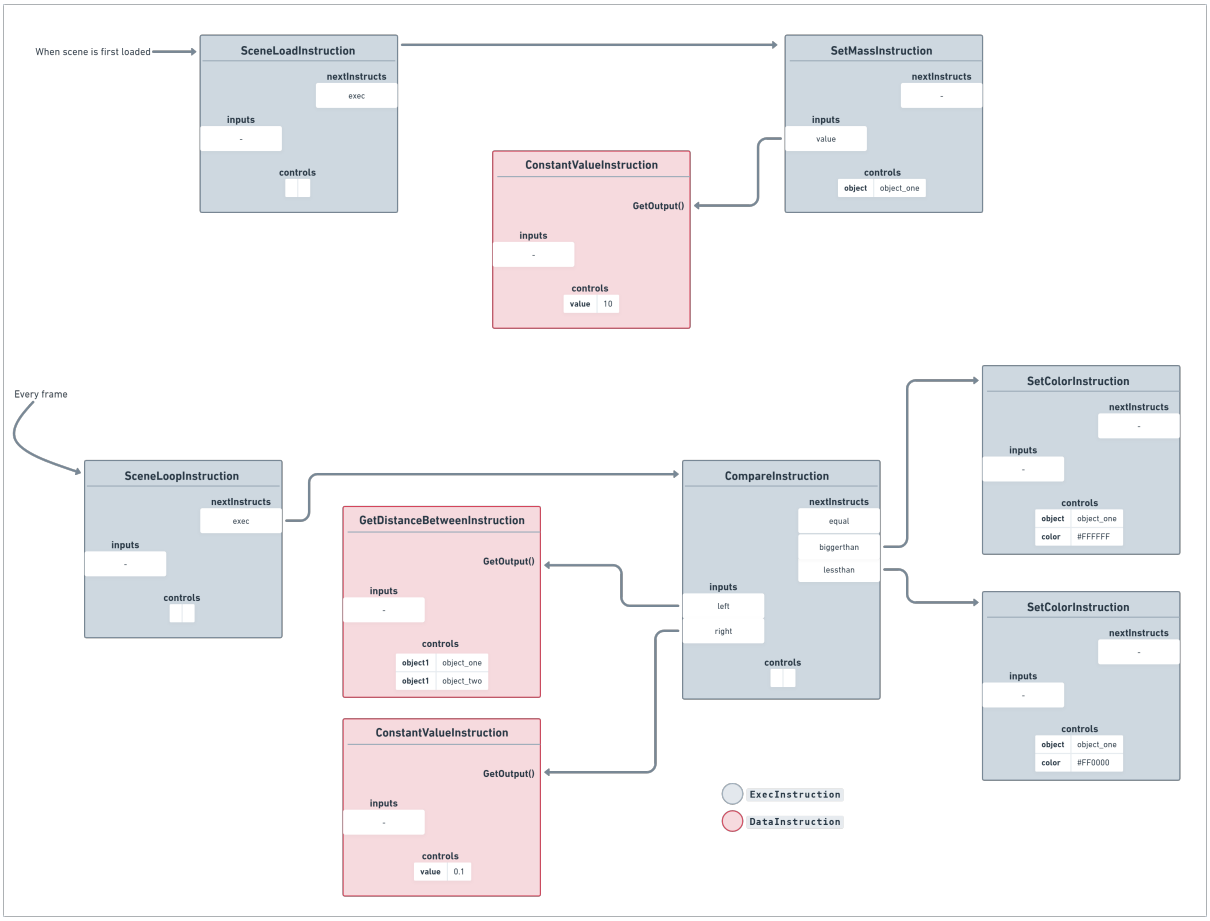


Figure 3.13: Example of logic data after being built into a graph structure (chain of responsibility)

The building process is done through a Depth First Search (DFS) algorithm that loops through all the nodes and processes every one. Some nodes are connected to other nodes and hence, a DFS occurs where nodes are processed as needed. Algorithm 1 shows the pseudocode of the DFS.

Algorithm 1 The algorithm used to convert the flattened structure into a graph.

```
1: _logicData ← the data from firstore.
2: cache ← {}
3: function GetInstructions
4:     for all logicItem ∈ _logicData do
5:         ConvertInstruction(logicItem.id, logicItem)
6:     end for
7:     return cache
8: end function
9: function ConvertInstruction(id, logicNode)
10:    if id ∈ cache then
11:        return cache[id]
12:    end if
13:    controls ← logicNode.controls
14:    inputs ← {}
15:    nextInstructs ← {}
16:    for all (key, value) ∈ logicNode.execOutputs do
17:        nextInstructs[key] ← ConvertInstruction(value, _logicData[value])
18:    end for
19:    for all (key, value) ∈ logicNode.inputValues do
20:        inst ← a ConstantValueInstruction with the value.
21:        inputs[key] ← new InputParam(inst, "value")
22:    end for
23:    for all (key, value) ∈ logicNode.inputsFrom do
24:        inst ← ConvertInstruction(value.nodeId, _logicData[value.nodeId])
25:        nextInstructs[key] ← new InputParam(inst, value.outputName)
26:    end for
27:    cache[id] ← An object that implements either DataInstruction or ExecInstruction
       is created.
28:    return cache[id]
29: end function
```

The algorithm briefly does the following:

1. A cache is checked to ensure that the current node is not already processed. If it was processed, simply return the instruction resulting from this node.

2. The initial values of `controls`, `inputs`, and `nextInstructs` are set. `controls` is very basic as the values are always fixed by the Teacher.

3. For every execution output, the nodes to which this node transitions, are processed into a `DataInstruction` *(the base class)* by calling the same function recursively. Then, the resulting instruction is saved into the `nextInstructs` dictionary.

4. Next, the inputs with fixed values set by the Teacher are processed. To simplify the process, a `ConstantValueInstruction` is created as a specialized type of `Data Instruction`. Its sole purpose is to return the fixed value specified by the Teacher. This instruction is then encapsulated within an `InputParam` object and saved in the `inputs` dictionary.

5. Then, the inputs that get their value from other `DataInstruction`s are processed. These are simply wrapped into an `InputParam` and then saved into the `inputs` dictionary.

6. Finally, an instruction is created based on the type of the node via a factory design pattern. The resultant instruction is saved into the cache to avoid recomputation and then returned.

3.5.8 Logic Manager

The logic manager is the component responsible for the execution of the logic of the scene. It also holds all the instructions after being built by the Logic Builder.

As seen in Figure 3.11, the `ExecInstruction` abstract class defines two properties called `IsStartInstruction` and `IsLoopInstruction` where the earlier property signifies that the instruction is a starting instruction that must be called on scene load while the latter signifies that the instruction must be called every frame. In this application, these properties are only overridden with True in the instructions `SceneLoadInstruction` and `SceneLoopInstruction`.

The class is initiated with a list of all the `SceneLoad` instructions as well as the `Scen eLoop` instructions. Thus when execution is started, all the `SceneLoad` are executed.

The `LogicManager` class is a unity component and thus has an `Update` event. This event is used to execute all the `SceneLoop` instructions in the scene in every frame.

Once a `SceneLoop` or `SceneLoad` instruction is executed, execution continues to the connected exec instructions and so on, in a "chain of responsibility" design pattern as seen in Figure 3.13. Some instructions require inputs and thus call other instructions in the middle of the path. The execution of the figure Figure 3.13 goes as follow:

- On scene start:
    1. `SceneLoadInstruction`
    2. `SetMassInstruction`
        (a) `ConstantValueInstruction` *(in order to get the fixed value of the mass)*
- Every scene frame:
    1. `SceneLoopInstruction`

2. `CompareInstruction`
   (a) `GetDistanceBetweenInstruction` *(gets the value of the left input)*
   (b) `ConstantValueInstruction` *(gets the value of the right input)*
   (c) One of the `SetColorInstruction`'s depending on the outcome of the `CompareInstruction`

Class Functions

1. `InitLogicManager(ExecInstruction[] startInstructs, ExecInstruction[] loopInstructs)`

   The main function of the class that inits it with the on load instructions and the on loop instructions.

2. `StartExecuting()`

   This method is called by SceneManager in order to execute the `SceneLoad` instructions. It also turns a flag on to start executing the `SceneLoop` instructions from the next frame and go on every frame.

3. `StopExecuting()`

   When called, the flag to execute the `SceneLoop` instructions every frame is turned off. Thus, the `SceneLoop` instructions for this scene are stopped from the next frame.

4. `FixedUpdate()`

   A function called by the Unity engine every frame which is used to execute the `SceneLoop` instructions every frame if and only if the flag is turned on.

3.5.9 Scene Lifetime

As mentioned earlier, the Scene Manager is the main driving component of the player application. When the app is first started, the experiment data is loaded via the `SceneLoader` and then built using the various builders. After that, the logic manager is started to handle the execution. A complete overview is in Figure 3.14. The scene goes as follows:

1. The scene manager calls the scene loader in order to load all the scene data from Firestore.
2. The first scene in the scene list is created by calling in SceneBuilder.
3. SceneBuilder uses the object builder in order to create the objects in the scene.
4. The object builder must download the 3D model. The model manager handles this job.
5. Model manager checks the local cache first, and if the model is not found, it is downloaded from Firebase storage.
6. The SceneBuilder now builds the scene logic using the DFS algorithm described in Algorithm 1.
7. Using the instructions built from the logic, the SceneManager calls the InitLogicManager function in the logic manager.
8. Finally, LogicManager starts the execution of the scene. Notice that the whole process starts again from step two when a different scene is loaded.

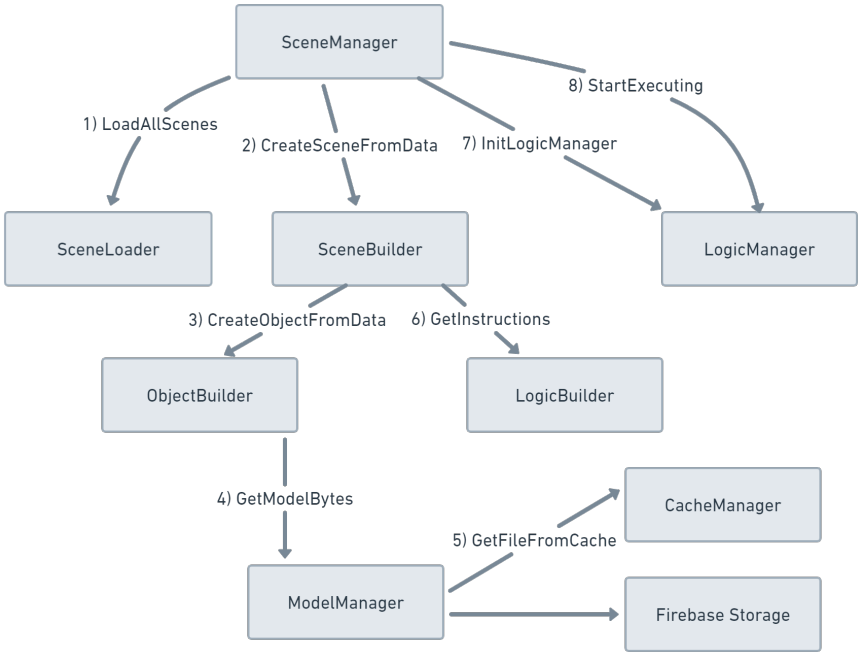


Figure 3.14: Scene loading lifetime

3.6 Experiment Examples

In this section, a few experiments designed using **EduAR** are presented. For every experiment, the design of it is presented as well as a brief explanation of the logic. A preview picture of the scene, running in the player, is also shown.

3.6.1 Convex Lens Experiment

This experiment is about showing how convex lenses form images. The lens has a focal point at which it changes behavior. When the object is moved by hand before the focal point, the image object is shown on the other side of the lens, magnified and upside down.

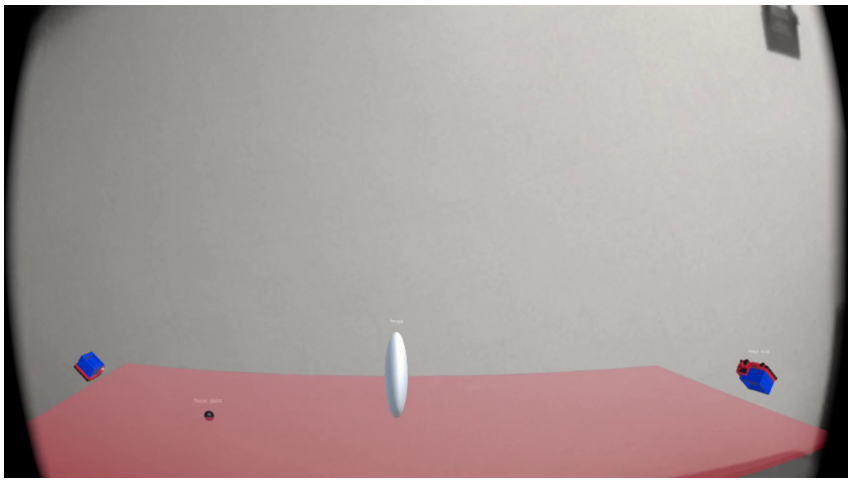


Figure 3.15: Lens Experiment: Truck placed before focal point

As the truck is moved between the focal point and the lens, the image shifts behind the lens and gradually aligns with the real truck at the center of the lens.

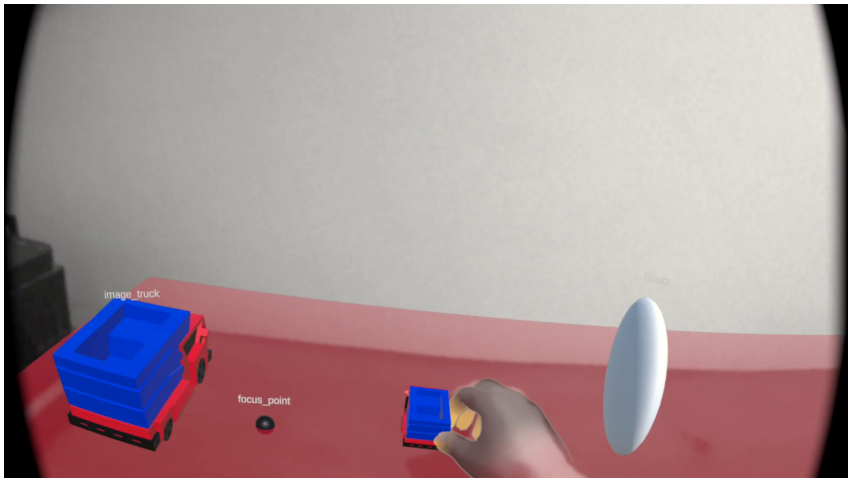


Figure 3.16: Lens Experiment: Truck placed after focal point

Experiment Implementation

The experiment contains four main objects:

1. focus_point: The point at which the focal point is defined. It is a black sphere in the scene.
2. orig truck: The truck that is moveable by the user.
3. lense: The lense object in the middle of the scene.
4. image_truck: The image of the truck, this object is not moveable by the player and is controlled entirely by the scene logic.

The scene logic is entirely in the `SceneLoop`. For every frame, the position of the real truck is compared with the position of the lens. If the distance is larger than the focal point then the position of the truck is set to $\frac{1}{1/3-1/a}$ where a is the distance between the lens and the truck. The rotation and scale are also updated with similar formulas. The `EvalInstruction` is used for the evaluation of these formulas.

When the distance between the lens and the truck is less than the focal point, the formulas are also applied, but reversed, as follows: $\frac{1}{1/a-1/3}$.

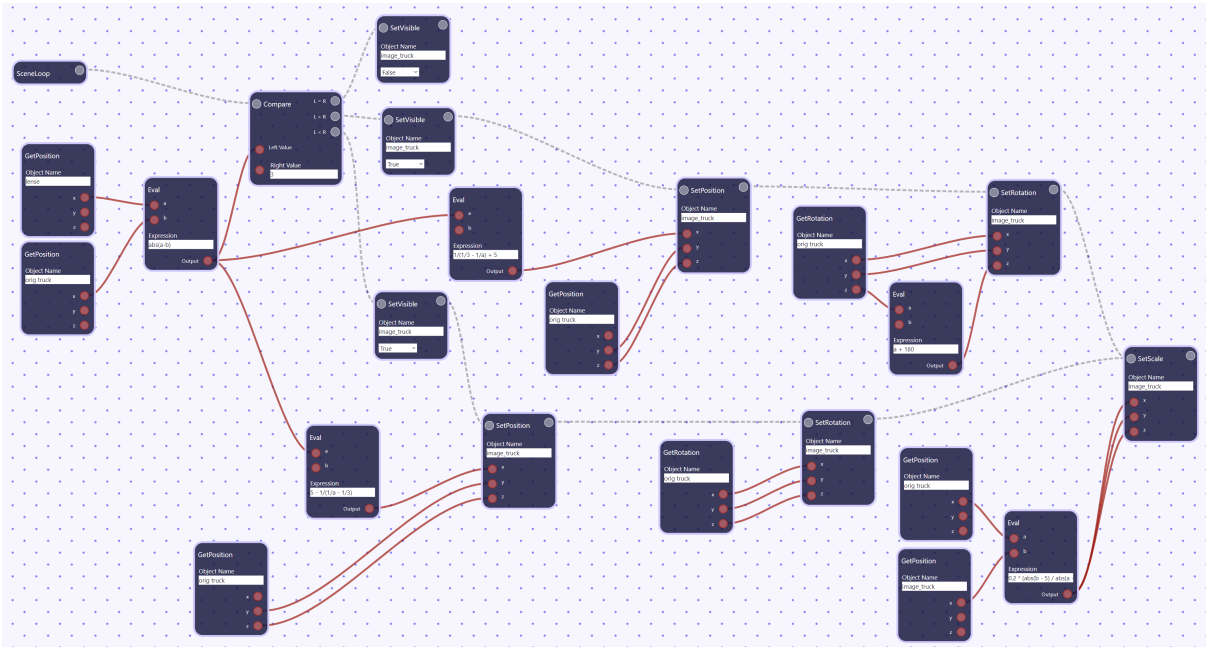


Figure 3.17: Lens Experiment: Scene Logic

3.6.2 Friction Experiment

The friction experiment aims to teach the user about the different friction types and how objects with different friction constants interact with each other. The experiment contains three cubes, three spheres and three platforms of three different materials: concrete, sand and ice. Each material has different friction values. For instance, an ice cube, placed on a concrete floor slides slowly, but slides fast when placed on the ice platform.

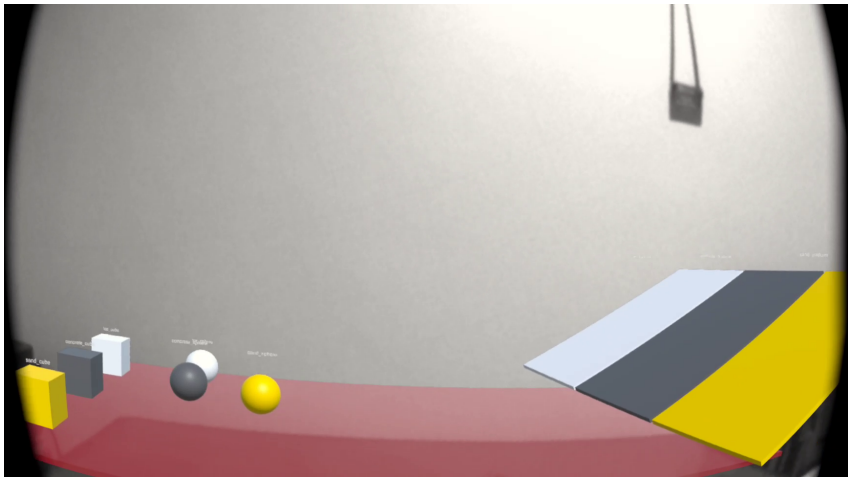


Figure 3.18: Friction Experiment: The platforms, cubes and spheres.

The platforms can be moved to change their angle of inclination by hand. Then, a sphere can be placed and it would slide. A cube of sand placed on a concrete platform would stick in place but would slide easily when placed on the ice platform.

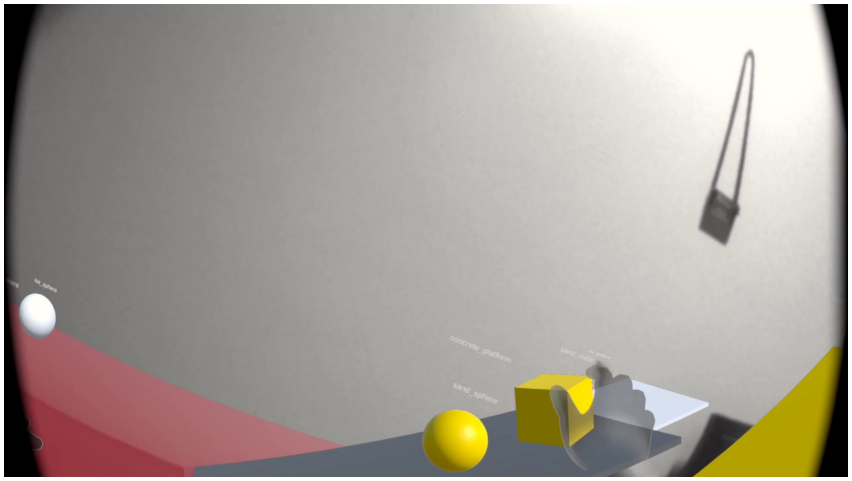


Figure 3.19: Friction Experiment: Sand cube and sphere placed on the concrete platform.

Experiment Implementation

The experiment contains nine objects where each three are of one material:

1. Sand objects:
   (a) A sphere (sand_sphere)
   (b) A cube (sand_cube)
   (c) A cube shaped as a platform (sand_platform)
2. Concrete objects:
   (a) A sphere (concrete_sphere)
   (b) A cube (concrete_cube)
   (c) A cube shaped as a platform (concrete_platform)
3. Ice objects: *Same as concrete and sand.*

The scene logic is simple. The friction is simulated using Unity's built-in physics engine. The logic has two parts. A part just for setting the friction constants to the correct values and another that fixes the platform's rotations in order to ensure the user can only rotate the platforms around their Z axis.

Figure 3.20: Friction Experiment: Scene Logic

3.6.3 Electricty Experiment

This experiment aims to teach the student about Ohm's law. The experiment contains a lamp, an ammeter and a voltmeter. The user has different resistors and batteries which can be attached to the circuit. The user can try different battery and resistor combinations and use the voltmeter and ammeter readings to determine the resistor of the lamp and verify Ohm's law.

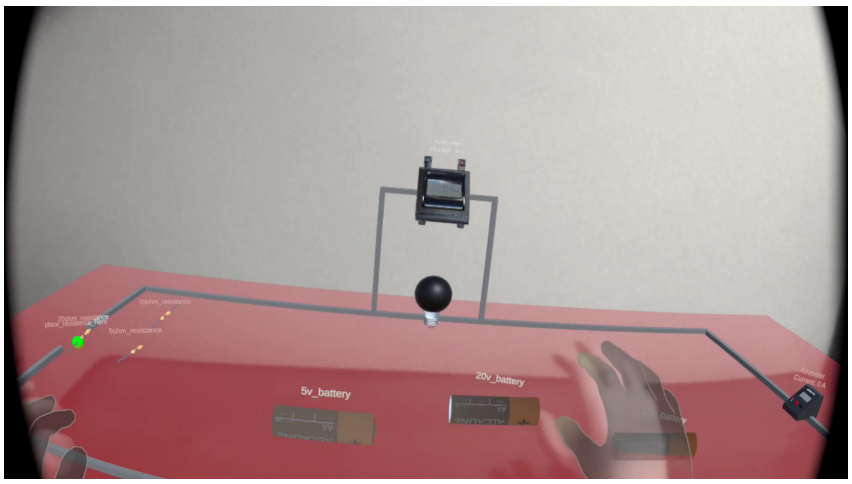


Figure 3.21: Electricty Experiment: Showing resistor attached but no energy source.

In addition, the lamp changes color depending if the circuit is closed or opened. The ammeter and voltmeter readings change instantly as well depending on which battery and resistor are connected.

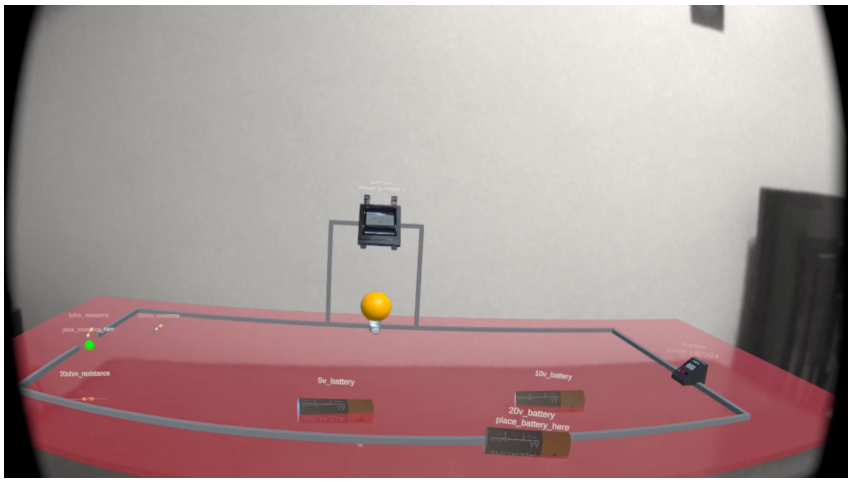


Figure 3.22: Electricty Experiment: Showing resistor and battery attached thus lamp is on.

Experiment Implementation

The scene contains the following objects:

1. Several cubes scaled, rotated and translated as needed in order to create the wires.
2. A voltmeter object.
3. An ammeter object.
4. Three resistor objects where each object has a different name representing the three different resistances.
5. Three battery objects where each object has a different name representing the three different voltages.

This scene's logic is divided into multiple parts. The first part involves setting the current voltage and resistance based on the distance between each voltage/resistor and its designated placement point. The values are saved into memory using `SetVariable`

Figure 3.23: Electricty Experiment: Scene Logic Part One

The second part is responsible for the calculation of the current. The value is calculated using an `Eval` block.

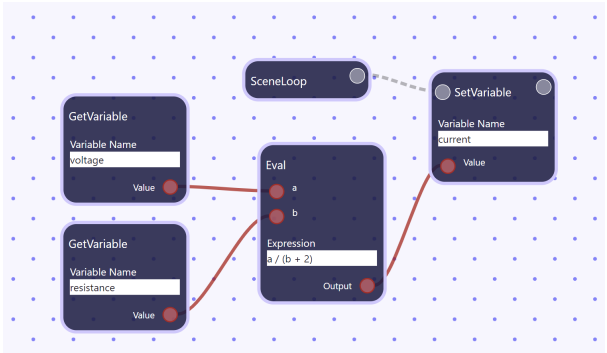


Figure 3.24: Electricty Experiment: Scene Logic Part Two

Finally, another part is responsible for the coloring of the lamp as well as the setting of the voltage and current values as the description of the voltmeter and ammeter respectively. The idea is that the descriptions appear on top of objects. An `EvalString` command is used to format the new description.

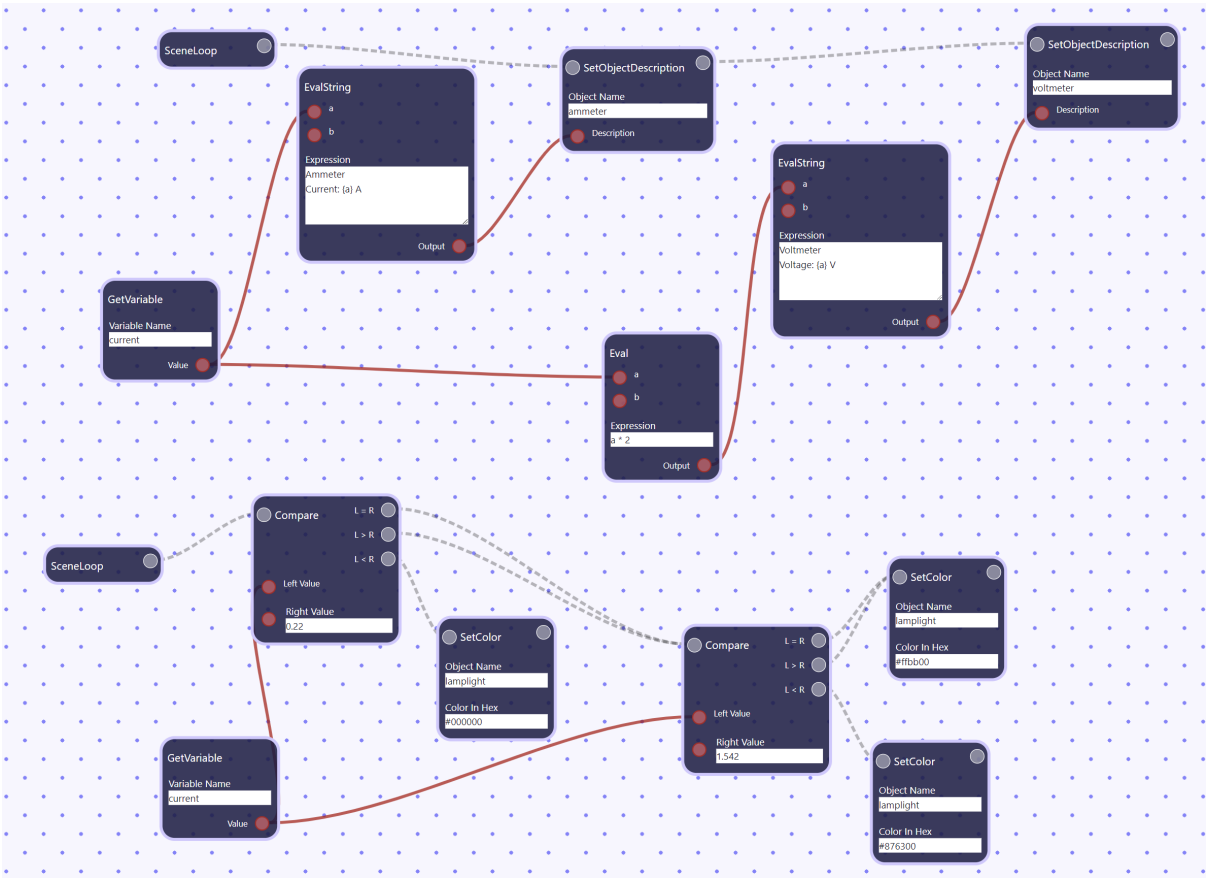


Figure 3.25: Electricty Experiment: Scene Logic Part Three

3.7 Summary

In summary, the system goals and a general overview of **EduAR** were presented. Then, both the designer application and the player were described from a user perspective showing the different user interfaces and how to interact with each of them. Following this, the database structure was defined. Subsequently, the implementation specifics of the designer were shown. Additionally, the player app implementation was also described. Finally, a few example experiments created using **EduAR** was shown.

Chapter 4

Conclusion

4.1 Achievements

The goal of the thesis was to develop an extensible and general AR experiments platform. A designer application was successfully built. The teacher uses the designer app to create their experiments. The experiment consists of the objects, their properties, and the logic of the scene. The logic is created via a drag-and-drop visual programming language. Then, the experiment data is successfully stored in Firebase. The user can then play the experiment on an Oculus headset. Once it is loaded, the user can start interacting with the scene. Examples of possible experiments can be seen in section 3.6.

4.2 Limitations

The main limitation of the thesis is proper testing and evaluation of the platform designed. It would be beneficial to test the platform with high school or university students and see how engaging and helpful the tool will be. Another evaluation factor is the simplicity and robustness of the designer app with the Teachers.

The app also does not have any error checking for the designer and invalid data placed into the logic blocks can cause crashes on the player side.

The AR technology in the Oculus Quest 2 is not mature yet and is limited to greyscale colors. However, the Quest Pro (subsection 2.2.2) headset and other similar technologies promise better screens and full colored passthrough. The technology will only improve.

4.3 Future Work

Study the system: The platform needs to be studied and analyzed on real users in order to know its effectiveness. It is suggested that a sample of students and teachers is selected and asked to try the system, then, their feedback would be analyzed.

User system: A login/registration system is yet to be added in order to separate the experiment data. Teachers should be able to sign up and have their own experiments and also be able to assign experiments to their students. The student would be able to see the experiments assigned to him once they are registered.

Add proper error checking: The logic designer in the designer application needs proper error checking to ensure the designs created will not crash the player side when executing.

Logic designer reusable components: The logic designer would be simpler if several widely used ideas are created into a reusable easier block. It would be also more beneficial if the Teacher can create reusable blocks themselves from the already existing blocks. An example would be to create an `ExecuteEvery` block that runs every x amount of seconds using the already existing blocks of get time, compare and set/get variables.

GPT4[24] Training: The fact that the database structure of the scene objects and logic is well defined (section 3.3.2) makes it possible to train a GPT4[24] model to autogenerate experiments based on a given prompt. Plugins could also be attached to create the 3D models. The idea would need to be further studied.

4.4 Summary

This thesis started by giving a background on AR and its impact on education, then the enabling technologies were discussed. Afterward, a review of already implemented platforms was discussed. After the discussion, a summary comparing the different platforms was shown. Moreover, the functionality of both the designer and the player applications was demonstrated before going into the different implementation details of each module of the system in both applications. The database structure was also described in detail. Finally, three example experiments were discussed and demonstrated in order to show what EduAR is capable of.

Appendix

Appendix A

Lists

AR	Augmented Reality
VR	Virtual Reality
LMS	Learning Management System
VPL	Visual Programming Language
DFS	Depth First Search

List of Algorithms

List of Figures

Bibliography

[1] Murat Akçayır and Gökçe Akçayır. "Advantages and challenges associated with augmented reality for education: A systematic review of the literature". In: *Educational Research Review* 20 (Feb. 2017), pp. 1–11. ISSN: 1747938X. DOI: 10.1016/j.edurev.2016.11.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S1747938X16300616 (visited on 03/03/2023).

[2] Ronald T Azuma. "A survey of augmented reality". In: *Presence: teleoperators & virtual environments* 6.4 (1997). Publisher: MIT press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . ., pp. 355–385.

[3] Mark Billinghurst and Hirokazu Kato. "Collaborative augmented reality". In: *Communications of the ACM* 45.7 (July 1, 2002), pp. 64–70. ISSN: 0001-0782. DOI: 10.1145/514236.514265. URL: https://doi.org/10.1145/514236.514265 (visited on 02/27/2023).

[4] *Blockly*. Google Developers. Mar. 4, 2023. URL: https://developers.google.com/blockly (visited on 03/04/2023).

[5] Inmaculada Coma-Tatay et al. "FI-AR learning: a web-based platform for augmented reality educational content". In: *Multimedia Tools and Applications* 78.5 (Mar. 2019), pp. 6093–6118. ISSN: 1380-7501, 1573-7721. DOI: 10.1007/s11042-018-6395-5. URL: http://link.springer.com/10.1007/s11042-018-6395-5 (visited on 02/24/2023).

[6] *Firebase*. Firebase. URL: https://firebase.google.com/ (visited on 05/10/2023).

[7] P. Grimm et al. "AMIRE - authoring mixed reality". In: *The First IEEE International Workshop Agumented Reality Toolkit,* The First IEEE International Workshop Agumented Reality Toolkit, Sept. 2002, 2 pp.–-. DOI: 10.1109/ART.2002.1107008.

[8] Gregory Kipper and Joseph Rampolla. *Augmented Reality: An Emerging Technologies Guide to AR*. Google-Books-ID: OyGiW2OYI8AC. Elsevier, Dec. 31, 2012. 177 pp. ISBN: 978-1-59749-734-3.

[9] *Kodu Game Lab — KoduGameLab*. URL: http://www.kodugamelab.com/ (visited on 05/30/2023).

[10] Keith M Lewin. *Science education in developing countries: Issues and perspectives for planners*. Citeseer, 1992.

[11] James Lin, Michael Thomsen, and James A Landay. "A Visual Language for Sketching Large and Complex Interactive Designs". In: 4 (2002).

[12] Moralejo Lucrecia et al. "AuthorAR: Authoring tool for building educational activities based on Augmented Reality". In: *2013 International Conference on Collaboration Technologies and Systems (CTS)*. 2013 International Conference on Collaboration Technologies and Systems (CTS). San Diego, CA, USA: IEEE, May 2013, pp. 503–507. ISBN: 978-1-4673-6404-1 978-1-4673-6403-4 978-1-4673-6402-7. DOI: 10.1109/CTS.2013.6567277. URL: http://ieeexplore.ieee.org/document/6567277/ (visited on 02/25/2023).

[13] Blair MacIntyre et al. "DART: A toolkit for rapid design exploration of augmented reality experiences". In: *Proceedings of the 17th annual ACM symposium on user interface software and technology*. UIST '04. Number of pages: 10 Place: Santa Fe, NM, USA. New York, NY, USA: Association for Computing Machinery, 2004, pp. 197–206. ISBN: 1-58113-957-8. DOI: 10.1145/1029632.1029669. URL: https://doi.org/10.1145/1029632.1029669.

[14] *Meta Quest 2: Immersive All-In-One VR Headset — Meta Store*. URL: https://www.meta.com/quest/products/quest-2/ (visited on 05/09/2023).

[15] Mark Newman et al. "DENIM: An Informal Web Site Design Tool Inspired by Observations of Practice". In: *Human-Computer Interaction* 18.3 (Sept. 1, 2003), pp. 259–324. ISSN: 0737-0024. DOI: 10.1207/S15327051HCI1803_3. URL: http://www.tandfonline.com/doi/abs/10.1207/S15327051HCI1803_3 (visited on 03/04/2023).

[16] Jong-Seung Park. "AR-Room: a rapid prototyping framework for augmented reality applications". In: *Multimedia Tools and Applications* 55.3 (Dec. 2011), pp. 725–746. ISSN: 1380-7501, 1573-7721. DOI: 10.1007/s11042-010-0592-1. URL: http://link.springer.com/10.1007/s11042-010-0592-1 (visited on 02/24/2023).

[17] Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. "Tips for creating a block language with blockly". In: *2017 IEEE Blocks and Beyond Workshop (B&B)*. 2017 IEEE Blocks and Beyond Workshop (B&B). Raleigh, NC: IEEE, Oct. 2017, pp. 21–24. ISBN: 978-1-5386-2480-7. DOI: 10.1109/BLOCKS.2017.8120404. URL: http://ieeexplore.ieee.org/document/8120404/ (visited on 03/04/2023).

[18] Plamen D. Petrov and Tatiana V. Atanasova. "The Effect of Augmented Reality on Students' Learning Performance in Stem Education". In: *Information* 11.4 (Apr. 15, 2020), p. 209. ISSN: 2078-2489. DOI: 10.3390/info11040209. URL: https://www.mdpi.com/2078-2489/11/4/209 (visited on 03/02/2023).

[19] *Rete.js - JavaScript framework for visual programming*. URL: https://retejs.org/ (visited on 05/17/2023).

[20] Hartmut Seichter, Julian Looser, and Mark Billinghurst. "ComposAR: An intuitive tool for authoring AR applications". In: *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*. ISMAR '08. USA: IEEE Computer Society, Sept. 15, 2008, pp. 177–178. ISBN: 978-1-4244-2840-3. DOI: 10.1109/ISMAR.2008.4637354. URL: https://doi.org/10.1109/ISMAR.2008.4637354 (visited on 02/24/2023).

[21] *Unity Technologies*. 2023. URL: https://unity.com/ (visited on 02/27/2023).

[22] Ana Villanueva et al. "Meta-AR-App: An Authoring Platform for Collaborative Augmented Reality in STEM Classrooms". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20: CHI Conference on Human Factors in Computing Systems. Honolulu HI USA: ACM, Apr. 21, 2020, pp. 1–14. ISBN: 978-1-4503-6708-0. DOI: 10.1145/3313831.3376146. URL: https://dl.acm.org/doi/10.1145/3313831.3376146 (visited on 02/20/2023).

[23] *Visual programming language*. In: *Wikipedia*. Page Version ID: 1142820697. Mar. 4, 2023. URL: https://en.wikipedia.org/w/index.php?title=Visual_programming_language&oldid=1142820697 (visited on 03/04/2023).

[24] Zezhong Wang et al. *Empower Large Language Model to Perform Better on Industrial Domain-Specific Question Answering*. May 19, 2023. arXiv: 2305.11541[cs]. URL: http://arxiv.org/abs/2305.11541 (visited on 05/27/2023).

[25] Steve Yuen, Gallayanee Yaoyuneyong, and Erik Johnson. "Augmented Reality: An Overview and Five Directions for AR in Education". In: *Journal of Educational Technology Development and Exchange (JETDE)* 4.1 (June 1, 2011). ISSN: 1941-8027. DOI: 10.18785/jetde.0401.10. URL: https://aquila.usm.edu/jetde/vol4/iss1/11.